SAFE PROGRAMMING AT THE C LEVEL OF ABSTRACTION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Daniel Joseph Grossman

August 2003

SAFE PROGRAMMING AT THE C LEVEL OF ABSTRACTION

Daniel Joseph Grossman, Ph.D.
Cornell University 2003

Memory safety and type safety are invaluable features for building robust software. However, most safe programming languages are at a high level of abstraction; programmers have little control over data representation and memory management. This control is one reason C remains the de facto standard for writing systems software or extending legacy systems already written in C. The Cyclone language aims to bring safety to C-style programming without sacrificing the programmer control necessary for low-level software. A combination of advanced compile-time techniques, run-time checks, and modern language features helps achieve this goal.

This dissertation focuses on the advanced compile-time techniques. A type system with *quantified types* and *effects* prevents incorrect type casts, dangling-pointer dereferences, and data races. An intraprocedural *flow analysis* prevents dereferencing NULL pointers and uninitialized memory, and extensions to it can prevent array-bounds violations and misused unions. Formal abstract machines and rigorous proofs demonstrate that these compile-time techniques are sound: The safety violations they address become impossible.

A less formal evaluation establishes two other design goals of equal importance. First, the language remains expressive. Although it rejects some safe programs, it permits many C idioms regarding generic code, manual memory management, lock-based synchronization, NULL-pointer checking, and data initialization. Second, the language represents a unified approach. A small collection of techniques addresses a range of problems, indicating that the problems are more alike than they originally seem.

BIOGRAPHICAL SKETCH

Dan Grossman was born on January 20, 1975 in St. Louis, Missouri. Thus far, his diligent support of St. Louis sports teams has produced one World Series victory in three appearances and one Super Bowl victory in two appearances, but no Stanley Cup finals. Upon graduating from Parkway Central High School in 1993, Dan's peers selected him as the male mostly likely to become a politician. Dan spent five summers working at the S-F Scout Ranch. For three years, he managed camp business operations with a typewriter, carbon paper, and an adding machine.

In 1997, Dan received a B.A. in Computer Science and a B.S. in Electrical Engineering from Rice University. These awards signified the termination of convenient access to La Mexicana, a restaurant that produces the world's best burritos. Also in 1997, Dan completed hiking the 2160-mile Appalachian Trail. To this day, he smirks when others describe somewhere as a, "really long walk."

For the last six years, Dan has lived primarily in Ithaca, New York while completing his doctorate in computer science at Cornell University. His ice-hockey skills have improved considerably.

Dan has been to every state in the United States except Alaska, Hawaii, Nevada, Michigan, Minnesota, Wisconsin, and South Carolina. (He has been to airports in Michigan, Minnesota, Nevada, and South Carolina.) In his lifetime, Dan has eaten only two green olives.

ACKNOWLEDGMENTS

Attempting to acknowledge those who have helped me in endeavors that have led to this dissertation is an activity doomed to failure. In my six years of graduate school and twenty-two years of formal education, far too many people have shared a kind word, an intellectual insight, or a pint of beer for this summary to be complete. But rest assured: I thank you all.

My advisor, Greg Morrisett, has been an irreplaceable source of advice, guidance, and encouragement. This dissertation clearly represents an outgrowth of his research vision. He has also shown an amazing ability to determine when my stubbornness is an asset and when it is a liability.

This dissertation owes its credibility to the actual Cyclone implementation, which is joint work with Greg, Trevor Jim, Michael Hicks, James Cheney, and Yanling Wang. These people have all made significant intellectual contributions as well as doing grunt work I am glad I did not have to do.

My thesis committee, Greg, Keshav Pingali, and Dexter Kozen (filling in for Jim West) have done a thorough and admirable job, especially considering this dissertation's length.

My research-group predecessors deserve accolades for serving as mentors, tutors, and collaborators. Moreover, they deserve individual mention for unique lessons such as the importance of morning coffee (Neal Glew), things to do in New Jersey (Fred Smith), fashion sense (Dave Walker), the joy of 8AM push-ups (Stephanie Weirich), and the proper placement of hyphens (Steve Zdancewic).

My housemates (fellow computer scientists Amanda Holland-Minkley, Nick Howe, and Kevin O'Neill) deserve mention for helping from beginning (when I didn't know where the grocery store was) to middle (when I couldn't get a date to save my life) to end (when I spent all my time deciding what to do next). My officemates (Neal, Steve Zdancewic, Yanling Wang, Stephen Chong, and Alexa Sharp) would only occasionally roll their eyes at a whiteboard full of $\vdash$ characters. (Not to mention an acknowledgments section with one.)

I have benefitted from the watchful eye of many faculty members at Cornell, most notably Bob Constable, Jon Kleinberg, Dexter Kozen, Lillian Lee, and An-

drew Myers. I also enjoyed great mentors during summer internships (John Reppy and Rob DeLine) and as an undergraduate (Matthias Felleisen). The Cornell Computer Science staff members, particularly Juanita Heyerman, have been a great help.

Friends who cooked me dinner, bought me beer, played hockey or soccer or softball, went to the theatre, played bridge, or in some cases all of the above, have made my time in Ithaca truly special. Only one such person gets to see her name in this paragraph, though. In an effort to be brief, let me just say that Kate Forester is responsible for me leaving Ithaca a happier person than I arrived.

My family has shown unwavering support in all my pursuits, so it should come as no surprise that they encouraged me throughout graduate school. They have occasionally even ignored my advice not to read my papers. But more importantly, they taught me the dedication, perseverance, and integrity that are prerequisites for an undertaking of this nature.

Finally, I am grateful for the financial support of a National Science Foundation Graduate Fellowship and an Intel Graduate Fellowship.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction and Thesis

Programming languages and their implementations are essential tools for software development because they provide a precise framework for specifying computer behavior and realizing the specification. Using a language is easier when the constructs of the language are at a level of abstraction suitable for the task at hand. The C programming language [132, 107, 123], originally developed for writing an operating system, has been used for just about every type of program. At the C level of abstraction, programs have almost complete control over the byte-level representation of data and the placement of that data in memory. Unlike at lower levels of abstraction, control flow is limited mostly to intraprocedural jumps and function call/return.

Hence the C programmer can manage her own data-processing and resource-management needs while forfeiting tedious assembly-level decisions such as instruction selection, register allocation, and procedure calling convention. This level of abstraction appeals for many tasks, such as operating systems, device drivers, (resource-constrained) embedded systems, runtime systems for higher level languages, data serializers (marshallers), etc. For lack of a better term, I call these problems *C-level tasks*.

Unlike many higher level languages, C does not provide enough strong abstractions to allow well-defined modular programs. For example, a bad fragment of a C program can arbitrarily modify any part of the entire program's data. Such incorrect behavior is much worse than a function that exits the program, diverges, or computes the wrong answer, because these are *local* well-defined effects.

To date, programmers of C-level tasks have had to choose between safe languages at higher levels of abstraction and unsafe languages at a more natural level. Language designers have proposed various solutions to this dilemma. First, convince developers that their task is not really a C-level task and that their desire to control data representation and resource management is misguided. Second,

provide debugging tools (traditional debuggers, "lint"-like tools, etc.) for the unsafe languages. Third, provide foreign-function interfaces so that safe-language code can call C code and vice-versa. Fourth, compile C code in an unconventional manner so that all safety violations can be detected when they occur at run time. As an alternative, I propose that *we can use a rich language of static invariants and source-level flow analysis to provide programmers a convenient safe language at the C level of abstraction.*

To substantiate this claim, colleagues and I have developed *Cyclone*, a programming language and implementation that is very much like C except that it is safe. This dissertation focuses on certain Cyclone language-design features. In particular, it evaluates Cyclone's type system and flow analysis, each of which addresses several safety issues in a uniform manner.

The rest of this introductory chapter further motivates compile-time guarantees for C-level programming (Section 1.1), provides a cursory description of the actual Cyclone language and this dissertation's narrower focus (Section 1.2), explains this dissertation's thesis (Section 1.3), highlights the technical contributions of this work (Section 1.4), and describes the structure of subsequent chapters (Section 1.5). I particularly urge reading Section 1.4 because it properly acknowledges others' work on Cyclone. This dissertation assumes familiarity with C, type systems, and operational semantics, but the first two chapters mostly require only familiarity with C.

1.1 Safe C-Level Programming

Memory safety is crucial for writing and reasoning about software. For example, consider a program that uses a simple password-checking routine like the following:

```
int check(char *p) {
  static char *pwd = "klff";
  return strcmp(p,pwd);
}
```

Because the `pwd` variable is visible only in the `check` function and this function never mutates `pwd`, we would like to conclude that the function always passes a pointer to an array holding `"klff"` as the second argument to `strcmp`. For a legal C program, this property holds.

However, there are many illegal C programs that C compilers do not reject. Although the implementation of such programs is undefined, conventional C compilers often choose implementations that could mutate `pwd`. Therefore, soundly reasoning about `check` requires that the rest of the program has no such *safety*

violations. This dissertation explains and prevents many safety violations, including incorrect type casts, dangling-pointer dereferences, data races, uninitialized memory, NULL-pointer dereferences, array-bounds violations, and incorrect use of unions.

Many safe languages exist, and they use a variety of techniques to enforce safety. Language restrictions can make certain violations impossible. For example, uninitialized memory is impossible if all declarations must have initializers. Automated memory management (garbage collection) prevents dangling-pointer dereferences. Advanced type systems can support generic code without allowing unsafe type casts. Run-time checks can prevent safety violations during execution. For example, most safe languages prevent array-bounds violations by storing array lengths with arrays and implementing subscript operations to check the lengths.

As a safe language, Cyclone uses many of these techniques. In particular, its use of quantified types is similar to ML [149, 40] and Haskell [130]. However, as a C-level language, it gives programmers substantial control over data representation, resource management, and the use of run-time checks. To do otherwise is to treat C as though it were a higher level language, which is counterproductive for C-level tasks. As such, it is inappropriate to rely exclusively on hidden fields (such as array lengths) and garbage collection. Instead, Cyclone programmers use the Cyclone language to express safety-critical properties, such as the lifetime of data objects and where array lengths are stored.

This design point is challenging: Compared to C, the language must express many properties that are exposed to programmers but cannot be described in the C language. Compared to higher-level languages, these properties are exposed to programmers rather than left to the implementation. This dissertation explores a set of uniform techniques that addresses this challenge.

1.2 Relation of This Dissertation to Cyclone

The Cyclone implementation is currently available on the World Wide Web at `http://www.cs.cornell.edu/projects/cyclone` and `http://www.research.att.com/projects/cyclone`. The distribution includes tens of thousands of lines of Cyclone code, in part because the compiler itself is written in Cyclone. An extensive user's manual [52] describes the full language and an overview has been previously published [126]. In this section, we briefly summarize Cyclone's techniques and applications before explaining this dissertation's focus and departures from actual Cyclone.

Cyclone is a safe programming language that retains (to the extent possible) the syntax, semantics, and idioms of C. Ideally, Cyclone would permit exactly

C programs that are safe, but it is well-known that this ideal is mathematically impossible. Therefore, we restrict programs to a more manageable subset of C, but such a subset by itself is too impoverished for realistic programming. Extensions discussed in detail in this dissertation let programmers express invariants that the Cyclone compiler would not otherwise infer. Other extensions capture idioms that would otherwise require C features disallowed in Cyclone. For example, Cyclone has exceptions but does not allow `setjmp` and `longjmp`.

In general, Cyclone ensures safety via a range of techniques including sophisticated types, intraprocedural flow analysis, run-time checking, and a safe interface to the C standard library. Preventing `NULL`-pointer dereferences provides a good example of how these techniques interact synergistically: The C library function `getc` has undefined (typically unsafe) behavior if callers pass it `NULL`. Rather than incur the run-time cost of checking for `NULL` in the body of `getc`, Cyclone's type for this function indicates that callers may not pass `NULL`. If an argument might not satisfy this precondition, Cyclone can insert a run-time check at the call site. Alternately, programmers can use the type system to propagate a not-`NULL` invariant through functions and data structures as appropriate. Furthermore, the flow analysis can often determine that extra checks are unnecessary because of conditional tests and loop guards in the source program.

Relying only on run-time checking would not let programmers control run-time cost or catch errors at compile-time. Relying only on invariants would prove too strong; some pointers are sometimes `NULL`. Relying only on intraprocedural flow analysis would prove too weak when safety relies on interprocedural invariants. By integrating these approaches, programmers can choose what is appropriate for their task without resorting to unsafe languages. Nonetheless, by implementing Cyclone like a conventional C implementation, programmers can easily resort to linking against C code. This ability makes Cyclone convenient for extending or incrementally porting systems already written in C.

Several projects have used Cyclone. First, my colleagues and I have used it to implement the Cyclone compiler and related tools (including a memory profiler, a documentation generator, a scanner generator, and a parser generator). We have also ported many C applications and benchmarks to Cyclone to measure the difficulty of porting and the run-time cost of ensuring safety [126]. I also ported a floppy-disk device driver for Windows to Cyclone. Encouragingly, almost the entire driver could be written in Cyclone. Discouragingly, there are ways to corrupt an operating system beyond the notions of safety that Cyclone captures, so the guarantees that Cyclone provides a device driver are necessary but insufficient. (In general, safety is always a necessary but insufficient aspect of correct software.)

Other researchers have used and extended Cyclone for several interesting systems. MediaNet [118] is a multimedia overlay network. Its servers are written in

Cyclone and exploit support for safe memory management. The Open Kernel Environment [27] allows partially trusted extensions in an operating-system kernel. They exploit the isolation that memory safety affords, but they employ additional run-time techniques to prevent excessive resource consumption. The RBClick [170] system uses a modified version of Cyclone for active-network extensions.

This dissertation focuses on Cyclone's type system and flow analysis. We ignore many important issues such as syntax, language extensions (such as exceptions and pattern matching), and idiosyncratic C features (such as variable-argument functions). We also ignore some safety-critical issues that are simple (such as preventing jumps into the scope of local variables) and difficult (such as supporting nul-terminated strings). We investigate neither the quantitative results nor the implementation experience cited above.

Rather, we focus on developing a core set of compile-time techniques that provides the foundation for Cyclone's safety. We explain these techniques, demonstrate their usefulness, develop abstract machines that model the relevant considerations, and prove (for the abstract machines) that the techniques work. As such, this dissertation is no substitute for the user's manual and does not serve as a primer for the language. Before Section 1.3 explains the thesis that these techniques demonstrate, we describe more specific disparities between actual Cyclone and this dissertation.

First, aspects of the language discussed in this dissertation are evolving; the discussion here may not accurately reflect the current language and implementation. For example, we are designing features that take advantage of restrictions on aliasing.

Second, this dissertation often deviates from Cyclone's concrete syntax in favor of more readable symbols, such as Greek letters. In general, compile-time variables are written like `‘a` (a back-quote character followed by an identifier), whereas I write α and allow subscripts (α_1) and primes (α').

Third, this dissertation ignores much of the difficulty in implementing Cyclone. For example, the implementation runs on several architectures and provides a safe interface to the C standard library, which was not designed with safety in mind. Another example is work to provide useful error messages despite Cyclone's advanced type system.

Fourth, the material in Chapters 5 and 7 has not been thoroughly implemented and tested. Although I am confident that the design described in these chapters is sound and useful, I cannot claim as much confidence as for the features that have been used extensively in the development of Cyclone.

1.3 Explanation of Thesis

With the previous section as background, I now explain what I mean by the thesis, *we can use a rich language of static invariants and source-level flow analysis to provide programmers a convenient safe language at the C level of abstraction.*

Rich Language of Static Invariants: The C type system describes terms with only enough detail for the compiler to generate code that properly accesses fields, calls functions, and so on. For the most part, a type in C is a size. Beyond size, C distinguishes floating-point types from other scalars and lets compilers adjust for alignment constraints precisely because code generation for conventional hardware requires doing so.

In contrast, the Cyclone type system is a much richer language. Types can distinguish the lifetime of an object, the length of an array, the lock that guards data, the value of an `int`, whether a pointer is `NULL`, and so on. These distinctions are crucial for preserving safety without resorting to compilation strategies and run-time checks inappropriate for the C level of abstraction. These additions describe invariants. For example, a pointer's type could indicate it refers to an array of at least three elements. An assignment can change which array it refers to, but it cannot cause it to refer to an array that is too short.

The Cyclone type system is not just an *ad hoc* collection of annotations. Each feature describes a safety-critical condition: An array must be some length; an object must be live; a lock must be held; a type equality must hold. Correspondingly, we have abstract compile-time variables for array lengths, object lifetimes, locks, and types. In fact, they are all just *type variables* of different *kinds*. As such, letting functions universally quantify over all kinds of compile-time variables is a natural feature that requires essentially no additional support for each kind. Similarly, we use tools like existential quantification, type constructors, effects, constraints, and singleton types more than twice. Subsequent chapters fully explain this jargon.

In short, by encoding the necessary safety conditions using well-understood type-system technology, we get a compile-time language that is rich and powerful yet uniform and elegant.

Source-Level Flow Analysis: For safety conditions for which invariants are too strong for effective programming, we use flow analysis to refine the static information for each program point. Examples include ensuring that programs initialize memory before using it and ensuring that an integer is less than an array bound. For an imperative language without implicit run-time checks, program-point specific information seems crucial. Although type theory can certainly describe such information, using a more conventional flow analysis appears more natural.

By *flow analysis*, I mean something more restrictive than just any analysis that ascribes different information to different program points. In particular, the analysis is *path insensitive*. For example, Cyclone rejects this program because the analysis concludes that `p` might be uninitialized:

```
int f(int x) {
  int *p;
  if(x)
    p = new 0;
  if(x)
    return *p;
}
```

At the point after the first conditional, we must assume `p` might be uninitialized. Because the return statement is reachable from this point, the analysis rejects the program. A more sophisticated analysis could determine that there are only two feasible execution paths in `f` and both are safe.

The distinction between flow-sensitivity and path-sensitivity actually depends on the domain of the analysis; that is, on the information we store at each program point. For example, if the analysis concludes that after the first conditional, "`p` is uninitialized only if `x` is `0`," it can conclude that the second conditional is safe.

Finally, the analysis is *source-level*, by which I mean its definition is in terms of Cyclone source programs and the compiler reports errors in source-level terms. This requirement is crucial for using a flow analysis as part of a language definition, as opposed to using it internally in a compiler. The distinction affects the design of the analysis. First, it leads me to favor simplicity even more than usual because the definition should be explainable (even if only language implementors truly understand the details). Second, it can make the analysis more difficult because we cannot define it in terms of a simpler intermediate language.

Convenient Safe Language for Programmers: A programming language should have a precise definition. So Cyclone is not just a tool that "magically" tries to find safety violations in C programs; it is a *language* with exact rules for what constitutes a legal program.

Cyclone is *safe*, an intuitive concept that is frustratingly difficult to define. Informally, we cannot write a Cyclone function that mutates the contents of an arbitrary address. More positively, parts of Cyclone programs can *enforce strong abstractions*. For example, consider this silly interface:

```
struct Foo; // an abstract type
struct Foo * make_foo(int);
int use_foo(struct Foo *);
```

Now consider this implementation:

```
struct Foo { int x; };
struct Foo * make_foo(int x) { return new Foo(2*x); }
int use_foo(struct Foo * s) { return s->x; }
```

In a safe language, we could conclude that the result of any call to `use_foo` is even (ignoring NULL pointers) because clients cannot break the `struct Foo` abstraction. In an unsafe language, poorly written or malicious clients could "forge" a `struct Foo` that held an `int` that `make_foo` did not compute.

It is actually trivial to define a safe language: Reject all programs. So one important aspect of *convenience* is allowing users to write the safe programs they wish to write. However, this definition is subject to the so-called "Turing Tarpit": Because almost all languages with loops or recursion are equally expressive (if you can write a program in one, there is some way to write an equivalent program in another), the ability to write a program with some behavior is an almost meaningless metric. In our case, a better goal is, "any safe C program is a legal Cyclone program." Because the safety of a C program is undecidable, we cannot attain this goal, but it remains a useful qualitative metric.

There are many possible answers to the question, "Does Cyclone accept this C program?" including:

- This unmodified C program is also a Cyclone program.
- This C program needs some Cyclone type annotations, but otherwise it is a Cyclone program.
- Some terms need local modification, but the overall structure of the program need not change.
- An equivalent Cyclone program exists, but it is necessary to change the data representation and control flow of the C program.

Roughly speaking, convenience favors the answers near the beginning of the list. For machine-generated programs, the difference between the first two answers is small; explicit type information increases the burden on *programmers*, so it is important to emphasize that Cyclone is designed for humans. Toward this end, certain decisions sacrifice expressiveness in favor of human convenience. The choice of *default annotations* is an important part of convenience for humans.

C Level of Abstraction: As noted at the beginning of this chapter, C differs from most higher level languages in that conventional implementations let the programmer guide the representation of data (e.g., the order of fields in a `struct` or the levels of indirection for a reference) and management of resources (e.g., reclamation of memory). This low-level control over data is important for C-level tasks; it is a primary reason that C remains a popular language for implementing low-level systems.

Strictly speaking, the C standard does not expose these representation and resource management details to programmers. An ANSI-C compliant implementation can add bounds fields to arrays, pad `struct` definitions, even check at run-time for dangling-pointer dereferences. In other words, one can implement C like a high-level language. But in doing so, one loses C's advantages for low-level systems. My thesis claims we can provide a safe C-like language without resorting to high-level implementation techniques. For example, the Cyclone implementation compiles pointers to machine addresses, just like conventional C compilers.

The *C Level of Abstraction* also distinguishes Cyclone from safe *lower* level languages, such as Typed Assembly Language [157]. Such languages require a level of detail appropriate for an assembly language, but C is often better than assembly for building large systems precisely because, in the interest of portability and programmer productivity, we are often willing to sacrifice control over details like calling convention, instruction selection, and register allocation.

Another measure of being C-level is the ease of interoperability with C itself. Because Cyclone does not change data representation or calling convention, programmers can give a C function a Cyclone type and call it directly from Cyclone code. There is no data-conversion cost, but the resulting program is only as safe as the Cyclone type the programmer chooses. For example, giving the C function `void * id(void*p) { return p; }` the type β `id(`α`)` "enriches" Cyclone with an unchecked cast. Nonetheless, if we can write almost all of a system in Cyclone, resorting to C only where necessary (much as one resorts to assembly where necessary in C applications), we can reduce the code that is subject to safety violations.

Rich type systems, flow analyses for safety, safe programming languages, and C-level abstractions are nothing new, but putting them together makes Cyclone a unique point in the language-design space. Bringing safety to a language aimed at helping develop low-level systems makes it possible to reason soundly about these systems in terms of user-defined abstractions. By focusing on compile-time techniques for safety, we can avoid performance costs and "hidden" run-time information. However, sound compile-time analysis is inherently conservative.

1.4 Contributions

Language design largely involves combining and adapting well-known features, so it can be difficult to identify original contributions beyond, "getting it all to work together." The related-work descriptions in subsequent chapters identify this dissertation's unique aspects (to the best of my knowledge); here I briefly discuss the "highlights" and where I have published them previously.

- The adaptation of quantified types to a C-like language is mostly a straightforward interpolation between higher level polymorphic languages with uniform data representation and Typed Assembly Language [157, 155], which in its instantiation for the IA-32 assembly language had a kind for every size of type. However, a subtle violation of type safety caused by a natural combination of mutation, aliasing, and existential types was previously unknown. I published the problem and the solutions explored in Chapter 3 in the 2002 European Symposium on Programming [94].

- The most novel aspects of the static type system for region-based memory management explored in Chapter 4 involve techniques for making it palatable in a source language without using interprocedural analysis. These aspects include the default annotations for function prototypes and the `regions(`α`)` operator for representing the region names of a type. Other contributions include integrating regions with conservative garbage collection, integrating regions with stack-allocated storage (though the Vault system [55] developed similar ideas concurrently), and subtyping based on the "region outlives" relationship (though the more dynamic RC compiler [86] has a similar notion). With others, I published a description of Cyclone memory management in the 2002 ACM Conference on Programming Language Design and Implementation [97].

- The type system for mutual exclusion in Chapter 5 adapts a line of work by others [73, 72, 74, 31, 29] aimed mostly at Java [92]. Nobody had adapted these ideas to a language with type variables before; a main contribution is realizing a striking correspondence between the solutions in Chapter 4 for regions and the solutions that seem natural for threads. Other contributions include a small extension that makes it easier to reuse code for thread-local and thread-shared data even if the code uses a "callee-locks" idiom, an integration with region-based memory management that does not require garbage collection for all thread-shared data, and a notion of "sharability" for enforcing that thread-local data remains thread local. I published this

work in the 2003 ACM International Workshop on Types in Language Design and Implementation [95].

- Using flow analysis to detect uninitialized memory or `NULL`-pointer dereferences is an old idea. Java [92] elevated the former to part of a source-language definition. The most interesting aspects of the analysis developed in Chapter 6 are its incorporation of must points-to information and its soundness despite under-specified order of evaluation. The definite-assignment analysis in Java is simpler because there are no pointers to uninitialized memory and Java completely specifies the order of evaluation.

- The singleton integer types in Chapter 7 for array bounds and discriminated unions are straightforward given the insights of the previous chapters. Having already provided compile-time variables of various kinds, addressed the interaction between polymorphism and features like mutation and nonuniform data representation, and developed a sound approach to flow analysis, checking certain integer equalities and inequalities proved mostly straightforward. The extensions to the flow analysis appear novel and interesting, but they are weaker than a sophisticated compile-time arithmetic like in DML [221].

Another important contribution under-emphasized in most of this dissertation is the Cyclone implementation, a joint effort with several others (see below). Together, we have written or ported well over 100,000 lines of Cyclone code. Type variables, regions, and definite assignment have proven crucial features in our development and I am confident that these aspects of Cyclone are "for real." Multithreading and singleton integers are more recent experimental features that remain largely unimplemented. In other words, the material in Chapters 3, 4, and 6 has been thoroughly exploited, unlike the material in Chapters 5 and 7.

As we will see consistently in this dissertation, potential aliasing is a primary cause of restrictions made to maintain safety. A powerful technique is to establish values must not be aliases, either via analysis ([158], Chapter 10) or an explicit type system [206, 202, 186, 55]. For the most part, Cyclone as described here has *not* taken this approach. Nonaliasing is an important tool for a safe expressive language, but this dissertation focuses on "how far we can go" without it. There is an important exception: We use the fact that when memory is allocated, there are no aliases to the memory.

Cyclone is a collaboration with a number of fabulous colleagues. Trevor Jim at AT&T Research and Greg Morrisett at Cornell University are Cyclone's original designers and they continue to be main designers and implementors as the language evolves. Many other people have contributed significantly to the design

and implementation, including Matthieu Baudet, James Cheney, Matthew Harris, Michael Hicks, Frances Spalding, and Yanling Wang.

It would be impossible to identify some particular feature of Cyclone and say, "I did that," for at least two reasons. First, language design is often about interactions among features, so designing a feature in isolation makes little sense. Second, the Cyclone team typically designs features after informal conversations and refines them after members of the team have experience with them. Nonetheless, this dissertation presents features for which I am mostly responsible. Subject to the above caveats, the work here is roughly my own with the following exceptions:

- Greg Morrisett designed Cyclone's type variables. I discovered the bad interaction with existential types (see Chapter 3), which (for obscure reasons) were not a problem in early versions of Cyclone.
- The formalism for regions in Chapter 4 is joint work with Greg Morrisett and Yanling Wang. Greg Morrisett implemented most of the type-checking to do with regions. Michael Hicks provided some of the examples and text in Section 4.1. Choosing the default annotations was a group effort.
- Greg Morrisett designed and implemented the compile-time arithmetic in Chapter 7 that enables nontrivial arithmetic expressions for array subscripts and union discrimination.

1.5 Overview

The next chapter provides a series of examples that explain the key ideas of this dissertation informally. Readers familiar with quantified types and flow analysis may wish to skip this description; it is not referred to explicitly in subsequent chapters. Conversely, readers wanting just the "rough idea" may wish to read Chapter 2 exclusively.

The next five chapters address different Cyclone features. In particular, Chapter 3 discusses type variables, Chapter 4 discusses the region system for memory management, Chapter 5 discusses multithreading, Chapter 6 discusses definite assignment and NULL pointers, and Chapter 7 discusses array bounds and discriminated unions. Chapters 3, 4, and 5 do not involve any flow analysis whereas Chapters 6 and 7 primarily involve flow analysis. Each chapter has a similar organization, with sections devoted to the following:

- A description of the safety violations prevented
- A basic description of the Cyclone features used to maintain safety while remaining expressive and convenient

- A more advanced description of the features, in particular how they interact with features from earlier chapters
- A discussion of limitations and how future work could address them
- A small formal language suitable for modeling the chapter's most interesting features
- A discussion of related work on the safety violations addressed

With the exception of Chapter 7, a rigorous proof establishes that each chapter's formal language has a relevant safety property. Because these proofs are long and detailed, I have relegated them to appendices. Appendices A, B, C, and D prove the theorems for Chapters 3, 4, 5, and 6 respectively. Each appendix begins with an overview of its proof's structure.

Understanding the main results of this dissertation should not require reading the sections on formal languages and the accompanying proofs. These languages add a level of precision not possible in English. Unlike full Cyclone, they allow us to focus on just some interesting features. The corresponding proofs are tedious, but they add assurance that Cyclone is safe and give insight about why Cyclone is safe. However, because the various chapters develop separate languages (related only informally via their similarity), it remains possible that some subtle interaction among separately modeled features remains unknown. Unfortunately, the syntactic proof techniques [219] that I use do not compose well because adding features often complicates many parts of the proofs. (Other techniques are ill-equipped to handle the complex models we consider.)

Chapter 8 discusses related work on safe C-like languages. Other projects have focused on techniques complementary to Cyclone's strengths, such as run-time checking and compile-time restrictions on aliasing. There are also many tools that sacrifice soundness in order to find bugs effectively without requiring as much explicit information from programmers.

Finally, Chapter 9 offers conclusions. First, I reiterate the ideas this chapter introduces about how a small set of techniques helps prevent a wide array of safety violations. The advantage of repeating this point later is that we can speak in terms of examples and technical details developed in the dissertation. Second, I discuss some general limitations of the approaches taken in this dissertation. I then briefly discuss some experience actually using Cyclone and place this work in the larger context of producing quality software.

Chapter 2

Examples and Techniques

To explain the safety violations endemic in C programs and how we can avoid them, we present a series of example programs. The Cyclone programs use a small set of techniques in several ways to address different safety violations. We use very simple examples to give the flavor of some interesting invariants and programs.

Example One: Bad Memory Access In simplest terms, this dissertation is about preventing programs like this one:

```
void f1() { *((int *)0xABC) = 123; }
```

A C compiler should accept this program. Technically, its meaning is undefined (implementation dependent), but programmers expect execution of `f1` to write `123` to address `0xABC` (or fail if the address is not writable). Because "address `0xABC`" is an assembly-language notion, understanding this program requires breaking any higher level abstraction of memory. For this reason, no code linked against code like `f1` can protect data, maintain invariants, or enforce abstractions. We have no desire to allow code like `f1`. Unfortunately, more reasonable C code can act like `f1` when it is used incorrectly.

2.1 Type Variables

Example Two: Type Equality for Parameters Many C programs assume the types of multiple values are the same, but the type system cannot state (much less enforce) this fact without choosing one particular type.

```
void f2(void **p, void *x) { *p = x; }
```

The function `f2` is a reasonable abstraction for assigning through a pointer, but type safety requires that `p` points to a value with the same type as `x`. Without this equality, a use of `f2` can violate memory safety:

```
int   y = 0;
int * z = &y;
f2(&z, 0xABC);
*z = 123;
```

The use of `f2` type-checks in C even though the first argument has type `int**` and the second argument has type `int`. C programmers would expect the call to assign `0xABC` to `z`.[1] Other functions with the same type, such as `f2ok`, could allow `&z` and `0xABC` as arguments:

```
void f2ok(void **p, void *x) { if(*p==x) printf("same"); }
```

Cyclone solves this problem with *type variables* and *parametric polymorphism*, much like higher-level languages including ML and Haskell. Programmers use them to state the necessary type equalities. For example, these examples are both legal Cyclone:

```
void f2(α *p, α x) { *p = x; }
void f2ok(α *p, β x) {}
```

Implicit in these examples is *universal quantification* over the free type variables α and β: The type of `f2` is roughly $\forall\alpha$. `void f2(`α`*, `α`)`. Uses of `f2` implicitly *instantiate* α. But in our example, no type for α would make `&z` and `0xABC` appropriate arguments. On the other hand, for `f2ok(&z,0xABC)`, it suffices to instantiate α with `int*` and β with `int`. Furthermore, we cannot give `f2` the type that `f2ok` has; the assignment in `f2` would not type-check.

In general, type variables let function types indicate what types must be the same while still allowing programs to apply functions to values of many types. To avoid needing code duplication or run-time type information, there are restrictions on what types can instantiate a type variable; we ignore this issue for now.

Example Three: Type Equality for First-Class Abstract Types To create a first-class data object with an abstract type, polymorphic functions do not suffice. A standard example is a *call-back*: A client registers with a server a call-back function together with data to pass to the call-back when invoking it. The server should allow different clients that use different types of data with their call-backs. In C, a simple version of this idiom could look like this:

[1] Technically, C does not guarantee that `sizeof(int)==sizeof(void*)`. We consistently ignore this detail.

```
struct IntCallback {
  int (*f) (void*);
  void *env;
};
struct IntCallback cb = {NULL,0};
void register_cb(void *ev, int fn(void*)) {
  cb.env = ev;
  cb.f   = fn;
}
int invoke_cb() { return cb.f(cb.env); }
```

Even if clients access `cb` via the two functions, the type of `register_cb` allows inconsistent types for the fields of `cb`:

```
int assign(int * x) {
  *x = 123;
  return 0;
}
void bad() {
  register_cb(0xABC, assign);
  invoke_cb();
}
```

As in the previous example, `void*` is too lenient to express the necessary type equalities: `invoke_cb` requires the parameter type of `cb.f` to be the same as the type of `cb.env`. The definition of `struct IntCallback` should express this requirement. Cyclone uses type variables and *existential quantification* [151]. For now, we present a simplified (incorrect) Cyclone program; we revise it in Chapter 4.

```
struct IntCallback { <α>
  int (*f)(α);
  α env;
};
struct IntCallback cb = {NULL,0};
void register_cb(α ev, int fn(α)) {
  cb = IntCallback(fn,env);
}
int invoke_cb() {
  let IntCallback{<β> fn, ev} = cb;
  fn(ev);
}
```

The type definition means that for any value of type `struct IntCallback`, *there exists* a type α such that the fields have the types indicated by the definition. The initializer for `cb` is well-typed by letting α be `int`. We call `int` the *witness type* for the *existential package* `cb`. The function `register_cb` changes the witness type of `cb` to the type of its parameter `ev`.

The bodies of `register_cb` and `invoke_cb` use special forms that make it easier for the type-checker to ensure that the functions use `cb` consistently. The expression form `IntCallback(fn,env)` is a "constructor expression"—it creates a value of type `struct IntCallback` with the first field holding `fn` and the second holding `env`. Initializing both fields in one expression makes it easy to check they use the same type for α. Assigning the fields separately leaves an intermediate state in which the necessary type equality does not hold. The declaration `let IntCallback{<`β`> fn,ev} = cb;` is a *pattern* that binds the type variable β and the term variables `fn` and `ev` in the following statement. The type variable gives an abstract name to the unknown witness type of `cb`; the pattern initializes `fn` and `ev` with `cb.f` and `cb.ev`, respectively. Extracting both fields at the same time ensures there is no intervening change in the witness type.

Example Four: Type Equality for Container Types Our final example of the importance of type variables is a fragment of a library for linked lists. In C, we could write:

```
struct List {
  void * hd;
  struct List * tl;
};
struct List * map(void * f(void *), struct List *lst) {
  if(lst==NULL)
    return NULL;
  struct List *ans = malloc(sizeof(struct List));
  ans->hd = f(lst->hd);
  ans->tl = map(f,lst->tl);
  return ans;
}
```

The function `map` returns a list that is the application of `f` to each element of `lst`. Type safety may require certain type equalities among the uses of `void*`. We intend for all `hd` fields in a linked list to hold values of the same type, but different lists may have values of different types. Furthermore, we expect `f`'s parameter to have the same type as `lst`'s elements, and we expect `f`'s result to have the same type as the elements in `map`'s result.

In Cyclone, we can express all these invariants:

```
struct List<α> {
  α hd;
  struct List<α> * tl;
};
struct List<β> * map(β f(α), struct List<α> * lst) {
  if(lst==NULL)
    return NULL;
  struct List<β> * ans = malloc(sizeof(struct List<β>));
  ans->hd = f(lst->hd);
  ans->tl = map(f,lst->tl);
  return ans;
}
```

Here `struct List` is a *type constructor* (i.e., a type-level function), not a type. So `struct List<int>` and `struct List<int*>` are different types. Nonetheless, polymorphism we can use `map` at either type provided the first argument is a function pointer of the correct type.

We have seen three common uses of `void*` in C, namely polymorphic functions, call-back types, and container types. Type variables let programmers express type equalities without committing to any particular type. Together with universal quantification, existential quantification, and type constructors, type variables capture so many uses of `void*` that Cyclone is a powerful C-like language without unchecked type casts.

These techniques are well-known in the theory of programming languages and in high-level languages such as ML and Haskell. Adapting the ideas to Cyclone was largely straightforward, but this dissertation explores some complications in great depth. Furthermore, many of the following examples show we can use these tools to capture static invariants beyond conventional types.

2.2 Singleton Integer Types

Oftentimes, C programs are safe only because `int` values are particular constants. By adding *singleton int* types and associated constructs, Cyclone lets programmers encode such invariants.

Example Five: Array-Bounds Parameters This C function is supposed to write `v` to the first `sz` elements of the array to which `arr` points:

```
void write_v(int v, unsigned sz, int *arr) {
  for(int i=0; i < sz; ++i)
    arr[i] = v;
}
```

To violate safety, clients can pass a value for `sz` greater than the length of `arr`. In Cyclone, pointer types include the bounds for the underlying array, but unlike languages such as Pascal, universal quantification lets us write functions that operate on arrays that have a length unknown to the callee:

```
void write_v(int v, tag_t<α> sz, int @{α} arr) {
  for(int i=0; i < sz; ++i)
    arr[i] = v;
}
```

In this example, α stands for an unknown *compile-time integer*, not a conventional type. The type of `arr` is now `int @{`α`}`, i.e., a not-`NULL` (that is what the `@` means) pointer to α many elements. The type `tag_t<`α`>` has only one value, the `int` that has the value of α. The distinction is a bit subtle: In this example, α is not a type; `tag_t<`α`>` is a type. In the following code, the type-checker accepts the first call, but rejects the second:

```
void f() {
  int x[256];
  write_v(0, 256, x);
  write_v(0, 257, x); // rejected
}
```

Example Six: Array-Bounds Fields When data structures refer to arrays, C programmers often use other fields to hold the array's size. In Cyclone, existential quantification captures this data-structure invariant, which we need to prevent bounds violations when using the `elts` field:

```
struct IntArr { <α>
  tag_t<α> sz;
  int @{α} elts;
};
void write_v_struct(int v, struct IntArr arr) {
  let IntArr{<β> s,e} = arr;
  write_v(v, s, e);
}
```

Another important idiom is discriminated unions: C programs that use the same memory for different types of data need casts or `union` types, but both are notoriously unsafe. However, it is common to use an `int` (or `enum`) field to record the type of data currently in the memory; this field *discriminates* which variant occupies the memory. Of course, programmers must correctly maintain and check the tag. By using singleton-int types (instead of `int`) and a richer form of `union` types, we can encode this idiom much like we encode array-bounds fields. Section 7.3 has examples.

We have not discussed how Cyclone ensures functions like `write_v` have safe implementations; that discussion is in Section 2.8. What we have discussed is how many of the same techniques—universal quantification, existential quantification, and type constructors—are useful for conventional types and integer constants. In higher level languages, language mechanisms such as bounded arrays and built-in discriminated unions make these advanced typing constructs less useful. By providing them in Cyclone, we impose fewer restrictions on data representation.

2.3 Region Variables

Another way to violate safety in C is to *dereference a dangling pointer*, i.e., access a data object after it has been deallocated. The access could cause a memory error ("segmentation fault"). More insidious, if the memory is reused (perhaps for a different type), the access could violate invariants of the new data object.

Example Seven: Dangling Stack Pointers The C compiler on my computer compiles this example such that a call to `g` attempts to write `123` to address `0xABC`.

```
int * f1() {
  int x = 0;
  return &x;
}
int ** f2() {
  int * y = 0;
  return &y;
}
void g() {
  int *  p1 = f1();
  int ** p2 = f2();
  *p1  = 0xABC;
  **p2 = 123;
}
```

The function `g` accesses the local storage for the calls to `f1` and `f2` after the storage is deallocated. Both calls use the *same* storage, so `p1` and `p2` become aliases even though they have different types. A C compiler can warn about such obvious examples as directly returning `&x`, but we can easily create equivalent examples that evade an implementation-dependent analysis.

In higher level languages, the standard solution to this safety violation is to give all (addressable) objects infinite lifetimes, conceptually. To avoid memory exhaustion, a garbage collector reclaims memory implicitly. In Cyclone, we want to manage memory like conventional C implementations (e.g., stack allocation of local variables) while preserving safety. Toward this goal, we partition memory into *regions*; all objects in a region have the same conceptual lifetime. Constructs that allocate memory (such as local-declaration blocks) have compile-time *region names* and pointer types include region names. The region name restricts where values of the type can point.

In our example, we cannot modify `f1` and `f2` to appease the type-checker because the return types would need to mention region names not in scope. (Chapter 4 describes this cryptic reason in detail. The point is that we use standard techniques—variables and scope—to help prohibit dangling-pointer dereferences.)

Requiring region names on all pointer types is not as restrictive or onerous as it seems due to universal quantification, type constructors, inference, and default annotations.

Example Eight: Region-Polymorphic Functions

```
int add_ps(int *ρ1 p1, int *ρ2 p2) { return *p1 + *p2; }
void assign(int *ρ1* pp, int *ρ1 p) { *pp = p; }
```

The function `add_ps` universally quantifies over region names ρ_1 and ρ_2; any two non-dangling pointers to `int` values are valid arguments. In fact, the type-checker fills in omitted region names on pointers in function parameters with fresh region names, so ρ_1 and ρ_2 are optional. Within function bodies, Cyclone infers region names. For these reasons, the earlier examples are correct (except as noted) despite omitted region names. In the function `assign`, we use the default rule to omit one region name, but we need the other two to establish that `p` has the type to which `pp` points. Without knowing this type equality, the assignment might later cause a dangling-pointer dereference after `p` is deallocated. For both functions, region polymorphism allows clients to call them with stack pointers, heap pointers, or a combination thereof.

Example Nine: Type Constructors With Region-Name Parameters

```
struct List<α,ρ> {
  α hd;
  struct List<α,ρ> *ρ tl;
};
```

The type constructor `struct List` now has two parameters, a type α for the elements and a region name ρ that describes where the "spine" of a list is allocated. Our earlier definition is legal Cyclone because unannotated pointers in type definitions default to a special heap region that conceptually lives forever. We can use our revised definition to describe lists with ever-living spines (by instantiating ρ with ρ_H, the name for the heap region) as well as lists that have shorter lifetimes (by instantiating ρ with some other region name).

We have not yet explained many other idioms, such as functions that return newly allocated memory. We have explained just enough to show how quantified types and type constructors help prove that programs do not dereference dangling pointers. Chapter 4 explains more advanced features and problems arising from the combination of regions and existential types.

2.4 Lock Variables

Multithreaded programs can use unsynchronized access to shared memory to violate safety. To discuss the problems, we assume a built-in function `spawn` that creates a thread that runs in parallel with the caller. The C prototype is:

```
void spawn(void (*f)(void *), void * arg, int sz);
```

The function `f` executes in a new thread of control. It is passed a pointer to a *copy* of `*arg` (or `NULL`). The third argument should be the size of `*arg`, which `spawn` needs to make a copy. The copy is *shallow*; the spawning and spawned thread share any memory reachable from `*arg`.

In Cyclone we write:

```
void spawn<α::AS>(void (@f)(α*), α* arg, sizeof_t<α> sz);
```

The `::AS` annotation indicates that it must be safe for multiple threads to share values of type α.

Example Ten: Pointer Race Condition On some architectures, concurrent access of the same memory location produces undefined results. This simple C program has such a potential *data race*:

```
int   g1 = 0;
int   g2 = 0;
int * gp = &g1;
void f1(int **x) { *x = &g2; }
int f2() { spawn(f1,&gp,sizeof(int*)); return *gp; }
```

If an invocation of `f2` reads `gp` while an invocation of `f1` writes `gp`, the read could produce an unpredictable bit-string. As we demonstrate below, Cyclone requires *mutual exclusion*—a well-known but sometimes too simplistic way to avoid data races—for accessing all thread-shared data (such as global variables).

Example Eleven: Existential-Package Race Condition On many architectures, we can assume that reads and writes of pointers are *atomic*; even without explicit synchronization, programs cannot corrupt pointer values. Even under this assumption, synchronization helps maintain user-defined data-structure invariants. Furthermore, it is necessary for safe mutable existential types, as this Cyclone code, which continues Example Three, demonstrates:[2]

```
void do_invoke(int *ignore) { invoke_cb(); }
int id(int x) { return x; }
void race(int * p) {
  register_cb(p,assign);
  spawn(do_invoke,NULL,sizeof(int));
  register_cb(0xABC,id);
}
```

The spawned thread invokes the call-back `cb`, which reads the two fields and calls one field on the other. Meanwhile, `race` uses `register_cb` to change `cb` to hold an `int` and a function expecting an `int`. A bad interleaving could have the spawned thread read the `f` field, then have the other thread change `cb`, and then have the spawned thread read the `env` field. In this case, we would expect the program to write to address `0xABC`. This situation arises because the two threads share the existential package.

Because of race conditions, multithreaded Cyclone requires all thread-shared data to be protected by a mutual-exclusion lock, or *mutex*. In order to let programmers describe which lock protects a particular thread-shared data object, we

[2]Recall that the code in Example Three is slightly incorrect because of memory management.

introduce *singleton lock types* and annotate pointer types with the lock that a thread must hold to dereference the pointer.

Example Twelve: Synchronized Access Universal quantification lets functions take locks and data that the locks guard, as this simple example shows:

```
int read(lock_t<ℓ> lk, int *ℓ x; {}) {
  sync lk { return x; }
}
```

The lock name ℓ is like a type variable, except it describes a lock instead of a type. The pointer type indicates that a thread must hold the lock named ℓ to dereference the pointer. The explicit *effect* `;{}` is necessary because the default effect for this function (see Chapter 5) would otherwise require the caller to hold the lock named ℓ. The term `sync` e s means, "acquire the lock e (blocking if another thread holds it), execute s, and release the lock."

Existential quantification allows storing locks in data structures along with data guarded by the locks. Type constructors with lock-name parameters allow a single lock to guard an aggregate data structure.

As shown above, pointer types for thread-shared data include a lock name; if the name is α, then a thread must hold a lock with type `lock_t<`α`>` to dereference the pointer. Thread-local pointers have a special annotation, much like ever-living data has a special region annotation. Thread-local data does not require synchronization.

In short, the basic system for ensuring mutual exclusion uses typing constructs very much like the memory-management system. Chapter 5 explains many issues regarding threads and locks, including:

- How to ensure code acquires a lock before accessing data guarded by the lock
- How to ensure thread-local data does not escape a single thread
- How to write libraries that can operate on thread-local or thread-shared data
- How to allow global variables in multithreaded programs

2.5 Summary of Type-Level Variables

C programs are often safe only because they maintain a collection of invariants that the C type system does not express. These invariants include type equalities among values of type `void*`, `int` values holding the lengths of arrays, `int` values

indicating the current variant of a `union` type, pointers not referring to deallocated storage, and mutual exclusion on thread-shared data. We have seen why these invariants are essential for safety.

To capture these idioms, Cyclone significantly enriches the C type system. In particular, we have added conventional type variables, singleton int constants, region names, and singleton lock names. Pointer types carry annotations that restrict the values of the type.

The important point is that these additions are uniform in the following sense. For each, we allow universal quantification, existential quantification, and type constructors parameterized by the addition. There are other similarities, such as how the type system approximates the set of live regions and the set of held locks, that we explain in Chapter 5.

However, these additions all enforce *invariants*; the type checker ensures some property always holds in a given, well-structured context. For local data, invariants are often too strong. We now give examples in which invariants are too strong. We use dataflow analysis in many such cases.

2.6 Definite Assignment

In C, we can allocate memory for a value of some type without putting a value in the memory. Using the memory as though a valid value were there violates safety.

Example Thirteen: Uninitialized Memory In this example, both assignment statements cause unpredictable behavior because of uninitialized memory.

```
void f() {
  int * p1;
  int ** p2 = malloc(sizeof(int*));
  *p1  = 123;
  **p2 = 123;
}
```

One simple solution requires programmers to specify initial values when allocating memory. For local declarations, initializers suffice. For heap memory, we provide the form `new` e, which is like `malloc` except that it initializes the new memory with the result of evaluating e. Another solution inserts initial values implicitly whenever programmers omit them. Doing so is difficult because of abstract types and separate compilation. It also violates the spirit of "acting like C."

These solutions, which make uninitialized memory impossible, ignore the fact that separating allocation from initialization is useful:

- Omitting an initializer serves as "self-documentation" that subsequent execution will initialize the value before using it.
- Correct C code is full of uninitialized memory because there is no `new` e; we would like to port such code to Cyclone without unnecessary modification.
- A common idiom is to stack-allocate storage for a value of an abstract type and then pass a local variable's address to an initializer (also known as a constructor) function. This idiom requires pointers to uninitialized memory.
- Initializing memory with values that the program will not use incurs unnecessary run-time cost.

In Cyclone, we allow uninitialized memory but check at compile-time that the program definitely assigns to the memory before using it. (The term *definite assignment* is from Java [92], which has a similar but less sophisticated flow analysis.) To do so, we maintain a conservative approximation of the possibly uninitialized memory locations for each program point.

Example Fourteen: Definite Assignment This simple example is correct Cyclone code:

```
int * f(bool b) {
  int *p1;
  if(b)
    p1 = new 17;
  else
    p1 = new 76;
  return p1;
}
```

This code is correct because no control-flow path to the return statement exists along which `p1` remains uninitialized. This example is simple for several reasons:

- The control flow is structured. In general, features like `goto` require us to analyze code iteratively.
- We have no pointers to uninitialized memory, such as with `malloc`.
- We have no under-specified order of evaluation (such as the order that arguments to a function are evaluated), which complicates having a sound, tractable analysis.

- We do not pass uninitialized memory to another function.

These complications (jumps, pointers, evaluation order, and function calls) are orthogonal to the actual problem (uninitialized memory), so we use one approach for all the problems we address with flow analysis. The essence of the approach is to incorporate *must points-to* information (e.g., "this pointer must hold the value returned by that call to `malloc`") into the analysis, and to require explicit annotations for interprocedural idioms like initializer functions.

2.7 NULL Pointers

The Cyclone type system distinguishes pointers that might be `NULL` (written τ`*` as in C) from those that are definitely not `NULL` (written τ`@`). Blithely dereferencing a τ`*` pointer with the `*` or `->` operators can violate safety.[3] One solution is for the compiler to insert an explicit check for `NULL` (throwing an exception on failure), but this check is often redundant, in which case the mandatory check introduces a performance cost.

Instead, we introduce checks only when our flow analysis cannot prove they are redundant. We can warn the user about inserted checks.

Example Fifteen: NULL Checks The compiler inserts only one check into this code:

```
int f(int *p, int *q, int **r) {
  int ans = 0;
  if(p == NULL) return 0;
  ans += *p;
  ans += *q; // inserted check
  *r = NULL;
  ans += *q;
}
```

The first addition needs no check because if `p` were `NULL`, the `return` statement would have executed. The last addition needs no check because if `q` were `NULL`, the second addition would have thrown an exception.

For sound reasoning about redundant checks, aliasing is crucial. For example, the last addition would need a check if `r` could be `&q`. The must points-to information addresses this need: A check involving some memory location is never eliminated if unknown pointers to the location may exist.

[3] Trapping access of address 0 (the normal implementation of `NULL`) is insufficient because `x->f` could access a large address.

2.8 Checking Against Tag Variables

Null-checks are easy to insert because the check needs only the pointer. For a subscript e_1`[`e_2`]` where e_1 has type τ`@{`α`}`, we must check that e_2 is (unsigned) less than α. To do so at run-time, we need a value of type `tag_t<`α`>`.[4] Implementations of safe high-level languages typically implement bounds-checking by storing such values in "hidden" locations. Doing so dictates data representation; it is a hallmark of high-level languages.

In Cyclone, we could pursue several alternatives. First, the implementation could try to find a value of type `tag_t<`α`>` in scope at the subscript. Doing so is awkward. Second, we could make subscript a ternary operator, forcing the programmer to provide the correct bound. This solution makes porting code more difficult and does not eliminate redundant tests.

The solution we actually pursue is to use the flow analysis in conjunction with the type system to prove that subscripts are safe. The main limitation is a restricted notion of mathematical equalities and inequalities. In this dissertation, I use only very limited notions (essentially equalities and inequalities between constants and variables) because the choice of a decidable arithmetic appears orthogonal to other issues.

Example Sixteen: Array-Bounds Checking In this example, the compiler accepts the first loop because the bound properly guards subscript. More formally, there is no control-flow path to `arr[i]` along which `i` might not be less than α. The compiler rejects the second loop because Cyclone includes no sophisticated compile-time arithmetic reasoning:

```
int twice_sum(tag_t<α> sz, int @{α} arr) {
  int ans=0;
  for(int i=0; i < sz; ++i)
    ans += arr[i];
  for(int j=1; j <= sz; ++j)
    ans += arr[j-1]; // rejected
  return ans;
}
```

Note that aliasing is still important. For example, if `i` were a global variable and the body of the first loop included a function call, the first loop might not be safe.

[4] Or something we know is smaller.

Example Seventeen: Implicit Checking Programmers who prefer the convenience of implicit checking can encode it with an auxiliary function:

```
struct MyArr<α> { <β>
  tag_t<β> sz;
  α @{β} elts;
}
α my_subscript(struct MyArr<α> arr, int ind) {
  let MyArr{<β> s, e } = arr;
  if(ind < s) return e[ind];
  throw ArrayBounds;
}
```

We can use the same techniques to check discriminated unions. In fact, the limited arithmetic is less draconian for `union` tags because the typical idioms (e.g., a `switch` statement) are easier to support.

2.9 Interprocedural Flow

We have seen how flow analysis can go beyond invariants to provide an expressive system for initializing memory, checking `NULL` pointers, checking array bounds, and checking `union` variants. But for scalability and separate compilation, we use *intraprocedural* flow analysis: For a function call, we make conservative assumptions based only on the function's type. We enrich function types with annotations that express flow properties. The compiler uses these properties to check the callee and the caller.

For example, if a function parameter is a pointer, we can say the function *initializes* the parameter. We check the function assuming the parameter points to uninitialized memory and we ensure that the function initializes the memory before it returns. At the call site, we allow passing a pointer to uninitialized memory and assume the function call initializes the memory.

For tag variables, we can express relations such as $\alpha \leq \beta$. Doing so shifts the burden of establishing the inequality to the caller (else the function call is rejected), allowing the callee to assume the relation. We can also introduce relations in type definitions: The creator of a value of the type must establish the relations. The user of a value can assume them.

Finally, we can consider not-null (`@`) types as shorthand for a property of possibly-`NULL` pointer types.

2.10 Summary of Flow-Analysis Applications

For properties that require multiple steps (e.g., allocation then initialization) or run-time checking (e.g., array bounds), a flow analysis proves valuable. It interacts with the type system synergistically: If the type system ensures that e_1 has type `tag_t<`α`>` and e_2 has type `tag_t<`β`>`, then given `if(`e_1`<=`e_2`)` s, the flow analysis can use $\alpha \leq \beta$ when checking s. Conversely, if s calls a function with a type requiring that one tag is less than another, then we can use the flow analysis to check the call.

Besides function calls and unstructured control flow, two features of Cyclone (and C) make a sound, tractable flow analysis technically interesting. The first is under-specified evaluation order. The second is rampant potential aliasing, even of local variables. The analysis in Chapter 6 makes conservative assumptions about these features. To remain effective, it incorporates must points-to information.

Chapter 3

Type Variables

Cyclone uses *type variables*, *quantified types*, and *type constructors* to eliminate the need for many potentially unsafe type casts in C while still allowing code to operate over values of different types. To begin, we review C's facility for casts and various idioms that are safe but require casts in C because of its impoverished type system. This discussion identifies the idioms that type variables capture.

Given an expression e of type `t1`, the C expression `(t2)`e *casts* e to type `t2`. At compile time, the expression `(t2)`e has type `t2`. At run time, it means the result of evaluating e is converted to a value type `t2`. The conversion that occurs depends on `t1` and `t2`.

If `t2` is a numeric type (`int`, `float`, `char`, etc.), the conversion produces some bit sequence that the program can then use as a number. The Cyclone type-checker allows such casts by conservatively assuming that any bit sequence might be the result. Casts to numeric types pose no problem for safety, so we have little more to say about them.

In C and Cyclone, neither `t1` nor `t2` can be an aggregate (`struct` or `union`) type[1] because it is not clear, in general, what conversion makes sense.

In C, programmers can cast an integral type (`int`, `char`, etc.) to a pointer type, but doing so is almost always bad practice. If a pointer of type `t1` is cast to integral type `t2` *and* `sizeof(t2)>=sizeof(t1)` *and* the resulting value is cast back to type `t1`, then we can expect to get the value of the original pointer. However, using `void*` is better practice, and Cyclone uses type variables in place of `void*`. So Cyclone forbids casting an integral type to a pointer type.

The only remaining casts are between two pointer types. One safe use of such casts is overcoming C's lack of *subtyping*. For example, given these type definitions, casting from `struct T1*` to `struct T2*` is safe:

[1] A `gcc` extension allows casting to `union U` if a field of `union U` has exactly type `t1`. This extension is not technically interesting.

```
struct T2 { int z; };
struct T1 { struct T2 x; int y; };
```

In C's implicit low-level view of memory, this cast makes sense because pointers are machine addresses and the first field of a `struct` begins at the same address as the `struct`. Cyclone allows these casts by defining the subtyping that is implicit in the C memory model and allowing casts only to supertypes. This dissertation does not describe subtyping in detail.

Another source of pointer-to-pointer casts is code reuse. If code manipulates only pointers and not the values pointed to, the code should work correctly for all pointer types. In C, the sanctioned way to write such *polymorphic* code is to use the type `void*` for the pointer types. To use polymorphic code, pointers are cast to `void*`. Presumably, some other code will eventually use the values pointed to. Doing so requires casting from `void*` back to the original pointer type. The safety problem is that nothing checks that this second cast is correct; a pointer of type `void*` could point to a value of any type. Cyclone forbids casting from `void*` to another pointer type, but does allow casting to `void*`.

The rest of this chapter explains how Cyclone's type variables eliminate most of the need for using `void*` by capturing the important idioms for code reuse. Another common use of `void*` is in user-defined discriminated unions; Chapter 7 explores that idiom in detail. Of course, determining if a C program casts from `void*` correctly is undecidable, so there exist correct C programs using `void*` that do not map naturally to Cyclone programs.

Section 3.1 presents how we use type variables and related features to describe programming idioms such as polymorphic code, first-class abstract types (e.g., function closures and call-backs), and libraries for container types. The material adapts well-known ideas to a C-like language; knowledgable readers willing to endure unusual syntax might skip it. Section 3.2 discusses how C's low-level memory model (particularly values having different sizes) complicates the addition of type variables. Section 3.3 discusses how type variables are safe in Cyclone despite mutation. It describes a newly discovered unsoundness involving aliased mutable existential types and Cyclone's solution. This section is the most novel in the chapter (although I previously published the idea [94]); it is important for language designers considering mutable existential types. Section 3.4 evaluates the type system mostly by describing its limitations. Section 3.5 presents a formal language for reasoning about the soundness of Cyclone's type variables, which is particularly important in light of Section 3.3's somewhat surprising result. Section 3.6 discusses related work. Appendix A proves type safety for the formal language.

3.1 Basic Constructs

One form of type in Cyclone is a *type variable* (written α, β, etc.). Certain constructs introduce type variables in a particular scope. Within that scope, the type variable describes values of an unknown type. The power of type variables (as opposed to `void*`) is that a type variable always describes the *same* unknown type, within some scope. We present each of the constructs that introduce type variables, motivate their inclusion, and explain their usage. We then present some techniques that render optional much of the cumbersome notation in the explanations. We defer complications such as nonuniform data representation to Section 3.2.

3.1.1 Universal Quantification

The simplest example of universal quantification is this function:

```
α id<α>(α x) { return x; }
```

This function is *polymorphic* because callers can *instantiate* α with different types to use the function for values of different types. For example, if `x` has type `int` and `y` has type `int*`, then `id<int>(x)` has type `int` and `id<int*>(y)` has type `int*`.

In general, a function can introduce *universally bound* type variables $\alpha_1, \alpha_2, \ldots$ by writing `<`$\alpha_1, \alpha_2, \ldots$`>` after the function name. The type variables' scope is the parameters, return type, and function body. The type of the function is a *universal type*. For example, the type of `id` is `α id<α>(α)`, pronounced, "for all α, `id` takes an α and returns an α." Using more conventional notation for universal types and function types, we would write $\forall\alpha.\alpha \to \alpha$. As Section 3.2 explains, `id` cannot actually be used for all types.

To use a polymorphic function (i.e., a value of universal type), we must *instantiate* the type variables with types. For example, `id<int>` has type `int id(int)`.

More interesting examples of polymorphic functions take function pointers as arguments. This code applies the same function to every element of an array of 10 elements.

```
void app10<α>(void f(α), α arr[10]) {
  for(int i=0; i < 10; ++i)
    f(arr[i]);
}
```

The function call type-checks because the argument has the type the function expects, namely α. To show that the code is reusable, we use it at two types:

```
int g; // global variable that functions modify
void add_int(int  x) { g += x;  }
void add_ptr(int *p) { g += *p; }
void add_intarr(int  arr[10]) { app10<int >(add_int, arr); }
void add_ptrarr(int* arr[10]) { app10<int*>(add_ptr, arr); }
```

We resorted to global variables only because the type of `app10`'s first argument let us pass only one argument (and, unlike in functional languages, we do not have function closures). A better approach passes another value to the function pointer. Because the type of this value is irrelevant to the implementation of `app10`, we make `app10` polymorphic over it.

```
void app10<α,β>(void f(β, α), β env, α arr[10]) {
  for(int i=0; i < 10; ++i)
    f(env,arr[i]);
}
int g; // global variable that functions modify
void add_int(int *p,  int  x)  { *p += x;  }
void add_ptr(int *p1, int *p2) { *p1 += *p2; }
void add_intarr(int  arr[10]) { app10<int*,int >(add_int,&g,arr) }
void add_ptrarr(int* arr[10]) { app10<int*,int*>(add_ptr,&g,arr) }
```

Now users of `app10` can use any pointer for identifying the value to modify, even one chosen based on run-time values.

In short, universal quantification over type variables is a powerful tool for encoding idioms in which code does not need to know certain types, but it does need to relate the types of multiple arguments (e.g., the array elements and the function-pointer's argument of `app10`) or arguments and results (e.g., the argument and return type of `id`). In C, we conflate all such types with `void*`, sacrificing the ability to detect inconsistencies with the type system.

In Cyclone, the refined information from polymorphism induces no run-time cost. Type instantiation is just a compile-time operation. The compiler does not duplicate code; there is one compiled version of `app10` regardless of the number of types for which the program uses it. Similarly, instantiation does not require the function body, so we can compile uses of `app10` separately from the implementation of `app10`.

We also do not use any run-time type information: We pass `app10` exactly the same information as we would in C. There are no "secret arguments" describing the type instantiation, which is important for two reasons. First, it meets our goal of "acting like C" and not introducing extra data and run-time cost. Writing reusable code is good practice; we do not want to penalize such code. Second, it

becomes complicated to compile polymorphic code differently than monomorphic code, as this example suggests:

```
α id<α>(α x) { return x; }
int f(int x) { return x+1; }
void g(bool b) {
  int (*g)(int) = (b ? id<int> : f);
  // use g
}
```

Because `id<int>` and `f` have the same type, we need to support (indirect) function calls where we do not know until run-time which we are calling. To do so without extra run-time cost, the two functions must have the same calling convention, which precludes one taking secret arguments and not the other.

Cyclone also supports *first-class polymorphism* and *polymorphic recursion*. The former means universal types can appear anywhere function types appear, not just in the types of top-level functions. This silly example requires this feature:

```
void f(void g<α>(α), int x, int *y) {
   g<int> (x);
   g<int*>(y);
}
```

Polymorphic recursion lets recursive function calls instantiate type variables differently than the outer call. Without this feature, within a function `f` quantifying over $\alpha_1, \alpha_2, \ldots$, all instantiations of `f` must be `f<`$\alpha_1, \alpha_2, \ldots$`>`. This silly example uses polymorphic recursion:

```
α slow_id<α>(α x, int n) {
  if(n >= 0)
    return *slow_id<α*>(&x, n-1);
  return x;
}
```

First-class polymorphism and polymorphic recursion are natural features. We emphasize their inclusion because they are often absent from languages, most notably ML, because they usually make full type inference undecidable [216, 114, 133]. Cyclone provides convenient mechanisms for eliding type information, but it does not support full inference. Therefore, it easily supports these more expressive features. We will find them more important in Chapters 4 and 5.

3.1.2 Existential Quantification

Cyclone `struct` types can *existentially quantify* over type variables, as in this example:

```
struct T { <α>
  α env;
  int (*f)(α);
};
```

In English, "given a value of type `struct T`, *there exists* a type α such that the `env` field has type α and the `f` field is a function expecting an argument of type α." The scope of α is the field definitions. A common use of such types is a library interface that lets clients register *call-backs* to execute when some event occurs. Different clients can register call-backs that use different types for α, which is more flexible than the library writer choosing a type that all call-backs process. When the library calls the `f` field of a `struct T` value, the only argument it can use is the `env` field of the *same* `struct` because it is the only value known to have the type the function expects. In short, we have a much stronger interface than using `void*` for the type of `env` and the argument type of `f`.

Existential types describe *first-class abstract types* [151]. For example, we can describe a simple abstraction for sets of integers with this type:

```
struct IntSet { <α>
  α elts;
  void (*add)(α,int);
  void (*remove)(α,int);
  bool (*is_member)(α,int);
}
```

The `elts` field stores the data necessary for implementing the operations. Abstraction demands that clients not assume any particular storage technique for `elts`; existential quantification ensures they do not. For example, we can create sets that store the elements in a linked list and other sets that store the elements in an array. The abstract types are *first-class* in the sense that we can choose which sort of set to make at run-time. We can even put sets using lists and sets using arrays together, such as in an array where each element has type `struct IntSet`. One cannot encode such data structures with universal quantification (and closed functions).

Most strongly typed languages do not have existential types *per se*. Rather, they have first-class function closures or first-class objects (in the sense of object-oriented programming). These features have well-known similarities with existential types. They all have types that do not constrain private state (fields of

existentially bound types, free variables of a first-class function, private fields of an object), which we can use to enforce strong abstractions. Indeed, a language without any such first-class data-hiding construct is impoverished, but any one suffices for encoding simple forms of the others. For example, we can use existential types to encode closures [150] and some forms of objects [33]. Many of the most difficult complications in Cyclone arise from existential types (we will have to modify the examples of this section in Chapter 4 and 5), but the problems would not disappear if we replaced them with another data-hiding feature. Providing no such feature would impoverish the language.

Cyclone provides existential types rather than closures or objects because they give programmers more control over data representation, which is one of our primary goals. Compiling closures or objects requires deciding how to represent the private state. Doing so involves space and time trade-offs that can depend on the program [7, 1], but programmers do not see the decisions. We prefer to provide a powerful type system in which programmers decide for themselves.

We now present the term-level constructs for creating and using values of existential types. We call such values *existential packages*. When creating an existential package, we must choose types for the existentially bound type variables, and the fields must have the right types for our choice. We call the types the *witness types* for the existential package. They serve a similar purpose to the types used to instantiate a polymorphic function. Witness types do not exist at run-time.

To simplify checking that programs create packages correctly, we require creating a package with a *constructor expression*, as in this example, which uses `struct T` as defined above:

```
int deref(int * x) { return *x;  }
int twice(int   x) { return 2*x; }
int g;
struct T makeT(bool b) {
  if(b)
    return T{<int*> .env=&g, .f=deref};
  return T{<int> .env=g, .f=twice};
}
```

If the code executes the body of the if-statement, we use `int*` for the witness type of the returned value, else we use `int`. The return type is just `struct T`; the witness type is not part of it. We never allow inconsistent fields: There is no τ such that `T{<`τ`> .env=g, .f=deref}` is well-typed.

To use an existential package, Cyclone provides *pattern matching* to *unpack* (often called *open*) the package, as in this example:

```
int useT(struct T pkg) {
  let T{<β> .env=e, .f=fn} = pkg;
  return fn(e);
}
```

The pattern binds `e` and `fn` to (copies of) the `env` and `f` fields of `pkg`. It also introduces the type variable β. The scope of β, `e`, and `fn` is the rest of the code block (in the example, the rest of the function). The types of `e` and `fn` are β and `int (*f)(`β`)`, respectively, so the call `fn(e)` type-checks. Within its scope, we can use β like any other type. For example, we could write β `x = id<`β`>(e);`.

We require reading the fields of a package with pattern matching (instead of using individual field projections), much as we require building a package all at once. For the most part, not allowing the "`.`" and "`->`" operators for existential types simplifies type-checking. When creating a package, we can check for the correct witness types. When using a package, it clearly defines the types of the fields and the scope of the introduced type variables. We can unpack a package more than once, but the unpacks will use different type variables (using the same name is irrelevant; the type system properly distinguishes each binding occurrence), so we could not use, for example, the function pointer from one unpack with the environment from the other.

3.1.3 Type Constructors

Type constructors with *type parameters* let us concisely describe families of types. Applying a type constructor produces a type. For example, we can use this type constructor to describe linked lists:

```
struct List<α> {
  α hd;
  struct List<α> * tl;
};
```

The type constructor `struct List` is a type-level function: Given a type, it produces a type. So the types `struct List<int>`, `struct List<int*>`, and `struct List<struct List<int>*>` are different. The type α is the formal parameter; its scope is the field definitions. Because the type of the `tl` field is `struct List<`α`>*`, all types that `struct List` produces describe homogeneous lists (i.e., all elements have the same type).

Type constructors can encode more sophisticated idioms. We can use this type constructor to describe lists where the elements alternate between two types:

```
struct ListAlt<α,β> {
  α hd;
  struct ListAlt<β,α> * tl;
};
```

Building and using values of types that type constructors produce is no different than for other types. For example, to make a `struct List<int>`, we put an `int` in the `hd` field and a `struct List<int>*` in the `tl` field. If `x` has type `struct List<int>`, then `x.hd` and `x.tl` have types `int` and `struct List<int>*`, respectively.

The conventional use of type constructors is to describe a "container type" and then write a library of polymorphic functions for the type. For example, these prototypes describe general routines for linked lists:

```
int length<α>(struct List<α>*);
bool cmp<α,β>(bool f(α,β), struct List<α>*, struct List<β>*);
struct List<α>* append<α>(struct List<α>*, struct List<α>*);
struct List<β>* map<α,β>(β f(α), struct List<α>*);
```

Compared to C, in which we would write just `struct List` and the `hd` field would have type `void*`, these prototypes express exactly what callers and callees need to know to ensure that list elements have the correct type. For example, for `append` (which we presume appends its inputs) to return a list where all elements have some type α, both inputs should be lists where all elements have this type. After calling `append` instantiated with type τ, the caller can process the result knowing that all elements have type τ.

Type constructors and existential quantification also interact well. For example, `struct Fn` is a type constructor for encoding function closures:

```
struct Fn<α,β> { <γ>
  β (*f)(γ, α);
  γ env;
};
```

This constructor describes functions from α to β with an environment of some abstract type γ. Of course, different values of type `struct Fn<`τ_1`,`τ_2`>` can have environments of different types. A library can provide polymorphic functions for operations on closures, such as creation, application, composition, currying, uncurrying, and so on.

Type constructors are extremely useful, but they cause few technical challenges in Cyclone. Therefore, the formalisms in this dissertation do not model them.

Parameters for `typedef` provide a related convenience. The parameters to a `typedef` are bound in the type definition. We must apply such a `typedef` to produce a type, as in this example:

```
typedef struct List<α> * list_t<α>;
```

The α to the right is the binding occurrence. Like in C, `typedef` is transparent: each use is completely equivalent to its definition. So writing `list_t<int>` is just an abbreviation for `struct List<int>*`.

3.1.4 Default Annotations

We have added universal quantification, existential quantification, and type constructors so that programmers can encode a large class of idioms for reusable code without resorting to unchecked casts. So far, we have focused on the type system's expressiveness without describing the features that reduce the burden on programmers. We now present these techniques and show how some of our examples require much less syntax. As we add more features in subsequent chapters, we revise these default rules to accommodate them.

First, in function definitions and function prototypes at the top-level (i.e., not within a function body or type definition), the outermost function implicitly universally quantifies over any free type variables. So instead of writing:

```
α id<α>(α x);
list_t<β> map<α,β>(β f(α), list_t<α>);
```

we can write:

```
α id(α x);
list_t<β> map(β f(α), list_t<α>);
```

Explicit quantification is still necessary for first-class polymorphism:

```
void f(void g<α>(α), int x, int *y);
```

Omitting the quantification would make `f` polymorphic instead of `g`.

Second, instantiation of polymorphic functions and selection of witness types can be implicit. The type-checker infers the correct instantiation or witness from the types of the arguments or field initializers, respectively. Some examples are:

```
struct T { <α> α env; int (*f)(α); };
struct T f(list_t<α> lst) {
  id(7);
  map(id,lst);
  return T{.env=7, .f=id};
}
```

Polymorphic recursion poses no problem because function types are explicit. Inference does not require immediately applying a function, as this example shows:

```
void f() {
  int (*idint)(int) = id;
  idint(7);
}
```

In fact, type inference uses *unification*, a well-known technique (see, e.g., [166]) not described in this dissertation, within function bodies such that all explicit type annotations are optional. Chapter 9 discusses some problems with type inference in Cyclone, but in practice we can omit most explicit types in function bodies. Every occurrence of a polymorphic function is implicitly instantiated; to delay the instantiation requires explicit syntax, as in this example:

```
void f(int x) {
  α (*idvar)<α>(α) = id<>; // do not instantiate yet
  idvar(x);  // instantiate with int
  idvar(&x); // instantiate with int*
}
```

Third, an unpack does not need to give explicit type variables. The type-checker can create the correct number of type variables and gives terms the appropriate types. We can write:

```
int useT(struct T pkg) {
  let T{.env=e, .f=fn} = pkg;
  return fn(e);
}
```

The type-checker creates a type variable β with the same scope that a user-provided type variable would have.

Fourth, we can omit explicit applications of type constructors or apply them to too few types. In function bodies, unification infers omitted arguments. In other cases (function prototypes, function argument types, etc.) the type-checker fills in omitted arguments with fresh type variables. So instead of writing:

```
int length(list_t<α>);
```

we can write:

```
int length(list_t);
```

In practice, we need explicit type variables only to express equalities (two or more terms have the same unknown type). There is no reason for the programmer to create type variables for types that occur only once, such as the element type for `length`, so the type-checker creates names and fills them in. We do not mean that type constructors are types, just that application can be implicit.

None of the rules for omitting explicit type annotations require the type-checker to perform interprocedural analysis. Every function has a complete type determined only from its prototype, not its body, so the type-checker can process each function body without reference to any other.

3.2 Size and Calling Convention

Different values in C and Cyclone can have different sizes, meaning they occupy different amounts of memory. For example, we expect a `struct` with three `int` fields to be larger than a `struct` with two `int` fields. Conventionally, all values of the same type have the same size, and we call the size of values of a type the size of the type. C implementations have some flexibility in choosing types' sizes (in order to accommodate architecture restrictions like native-word size and alignment constraints), but sizes are compile-time constants.

However, not all sizes are known everywhere because C has *abstract* `struct` declarations (also known as incomplete structs), such as "`struct T;`". To enable efficient code generation, C greatly restricts where such types can appear. For example, if `struct T` is abstract, C forbids this declaration:

```
struct T2 {
  struct T x;
  int y;
};
```

The implementation would not know how much room to allocate for a variable of type `struct T2` (or `struct T`). If `s` has type `struct T2*`, there is no simple, efficient way to compile `s->y`. In short, because the size of abstract types is not known, C permits only pointers to them.

In Cyclone, type variables are abstract types, so we confront the same problems. Cyclone provides two solutions, which we explain after introducing the *kind* system that describes them. Kinds classify types just like types classify terms. In this chapter, we have only two kinds, `A` (for "any") and `B` (for "boxed").[2] Every type has kind `A`. Pointer types and `int` also have kind `B`. We consistently assume, unlike

[2]I consider the term "boxed" a strange historical accident. In this dissertation, it means, "pointers and things represented just like them."

C, that `int` has the same size and calling convention as `void*`. Saying `int` is just more concise than saying, "an integral type represented like `void*`."

The two solutions for type variables correspond to type variables of kind `B` and type variables of kind `A`. A type variable's binding occurrence usually specifies its kind with α:`B` or α:`A`, and the default is `B`.[3] All of the examples in Section 3.1 used type variables of kind `B`. Simple rules dictate how the type-checker uses kinds to restrict programs:

- A universally quantified type variable of kind `B` can be instantiated only with a type of kind `B`.
- An existentially quantified type variable of kind `B` can have witness types only of kind `B`.
- If α has kind `A`, then α is subject to the same restrictions as abstract `struct` types in C. Essentially, it must occur directly under pointers and programs cannot dereference pointers of type α`*`.
- The type variables introduced in an existential unpack do not specify kinds. Instead, the i^{th} type variable has the same kind as the i^{th} existentially quantified type variable in the type of the package unpacked.

Less formally, type variables of kind `B` stand for types that we can convert to `void*` in C. That makes sense because all of the examples in Section 3.1 use type variables in place of `void*`. We forbid instantiating such an α with a `struct` type for the same reasons C forbids casting a `struct` type to `void*`. Type variables of kind `A` are less common because of the restrictions on their use, but here is a silly example:

```
struct T1<α:A> { α **x; α **y; };
void swap<α:A>(struct T1<α> *p) {
  α * tmp = *x;
  *x = *y;
  *y = tmp;
}
```

Because `swap` quantifies over a type of kind `A`, we can instantiate `swap` with any type.

A final addition makes type variables of kind `A` more useful. We use the unary type constructor `sizeof_t` to describe the size of a type: The only value of type

[3] In Cyclone, the default kind is sometimes `A`, depending on how the type variable is used, but we use simpler default rules in this dissertation.

`sizeof_t<`τ`>` is `sizeof(`τ`)`. As in C, we allow `sizeof(`τ`)` only where the compiler knows the size of τ, i.e., all abstract types are under pointers.

The purpose of `sizeof_t` is to give Cyclone types to some primitive library routines we can write in C, such as this function for copying memory:

```
void mem_copy<α:A>(α* dest, α* src, sizeof_t<α> sz);
```

Disappointingly, it is not possible to implement this function in Cyclone, but we can provide a safe interface to a C implementation. A more sophisticated version of this example appears in Chapter 7.

Not giving `float` kind `B` deserves explanation because we could assume that `float` has the same size as `void*`, as we did with `int`. Many architectures use a different calling convention for floating-point function arguments. If `float` had kind `B`, then we could not have one implementation of a polymorphic function while using native calling conventions, as this example demonstrates:

```
float f1(float x) { return x; }
α f2(α x) { return x; }
void f3(bool b) {
  float (*f)(float) = b ? f1 : f2;
  f(0.0);
}
```

As discussed in Section 3.6, the ML community has explored all reasonable solutions for giving `float` kind `B`. None preserve data representation (a `float` being just a floating-point number and a function being just a code pointer) without secret arguments or a possibly exponential increase in the amount of compiled code. In Cyclone, we prefer to expose this problem to programmers; they can encode any of the solutions manually.

3.3 Mutation

Type safety demands that the expressiveness gained with type variables not allow a program to view a data object at the wrong type. Mutable locations (as are common in Cyclone) are a notorious source of mistakes in safe-language design. In this section, we describe the potential pitfalls and how Cyclone avoids them.

3.3.1 Polymorphic References

Cyclone does not have so-called *polymorphic references*, which would allow programs like the following:

```
void bad(int *p) {
  ∀α.(α*) x = NULL<>; // not legal Cyclone
  x<int*> = &p;
  *(x<int>) = 0xABC;
  *p = 123;
}
```

We can give `NULL` any pointer type, so it is tempting to give it type $\forall\alpha.(\alpha*)$. By not instantiating it, we can give `x` the same type. But by assigning to an instantiation of `x` (i.e., `x<int*> = &p`), we put a nonpolymorphic value in `x`. Hence, the second instantiation (`x<int>`) is wrong and leads to a violation of memory safety.

To avoid this problem, it suffices to disallow type instantiation as a form of "left expression." (In C and Cyclone, the left side of an assignment and the argument to the address-of operator must be valid left expressions.) The formal languages in this dissertation use precisely this solution: $e[\tau]$ (the formal syntax for instantiation) is never a left expression. In fact, there are no values of types like $\forall\alpha(\alpha*)$ because the only terms with universal types are functions and functions are not left expressions. Most of the formal languages do not have `NULL`.

The solution in the actual Cyclone implementation is more convoluted because C does not have first-class functions (a function definition is not an expression form). Instead, using a *function designator* (some `f` where `f` is the name of a function) implicitly means `&f` and a function call implicitly dereferences the function pointer. In Cyclone, that means we must allow `&(f<`τ`>)` because that is what `f<`τ`>` actually means. No unsoundness results because code is immutable. Having code pointers of different types refer to the same code is no problem because none of the pointer types can become wrong. Expressions like `f<`τ`> = g` make no sense.

Another quirk allows the implementation not to check explicitly that left expressions of the form `e<`τ`>` have only function designators (or more type instantiations) for e: there is no syntax (concrete or abstract) for writing a universal type like $\forall\alpha.\tau$ unless τ is a universal type or a function type. Hence all type instantiations are ultimately applied to function designators.

In our formal languages, we do not use this quirk. The type syntax is orthogonal (e.g., $\forall\alpha.(\alpha*)$ is a well-formed type) even though all polymorphic values are functions. We also disallow $\&f$ where f is a function (definition). Instead, we must assign a function to a location and take the location's address. If we had a notion of immutability, we could allow $\&(e[\tau])$ as a left expression when e was a valid immutable left expression.

Section 3.6 briefly describes how other safe polymorphic languages prevent polymorphic references.

3.3.2 Mutable Existential Packages

It does not appear that other researchers have carefully studied the interaction of existential types with features like mutation and C's address-of (&) operator. Orthogonality suggests that existential types in Cyclone should permit mutation and acquiring the address of fields, just as ordinary `struct` types do. Moreover, such abilities are genuinely useful. For example, a server accepting call-backs can use mutation to reuse the same memory for different call-backs that expect data of different types. Using & to introduce aliasing is also useful. As a small example, given a value `v` of type `struct T {<`α`> `α` x; `α` y;};` and a polymorphic function `void swap(`β`*, `β`*)` for swapping two locations' contents, we would like to permit a call like `swap(&v.x, &v.y)`. Unfortunately, these features can create a subtle unsoundness.

The first feature—mutating a location holding a package to hold a different package with a different witness type—is supported naturally. After all, if `p1` and `p2` both have type `struct T`, then, as in C, `p1=p2` copies the fields of `p1` into the fields of `p2`. Note that the assignment can *change* `p2`'s witness type, as in this example:

```
struct T {<α> void (*f)(int, α); α env;};
void ignore(int x, int y) {}
void assign(int x, int *y) { *y = x; }
void f(int* ptr) {
  struct T p1 = T(ignore, 0xABC);
  struct T p2 = T(assign, ptr);
  p2 = p1;
}
```

Because we forbid access to existential-package fields with the "`.`" or "`->`" operators, we do not yet have a way to acquire the address of a package field. We need this feature for the `swap` example above. To use pattern matching to acquire field addresses, Cyclone provides *reference patterns*: The pattern `*id` matches any location and binds `id` to the location's address.[4] Continuing our example, we could use a reference pattern pointlessly:

```
  let T{<β> .f=g, .env=*arg} = p2;
  g(37,*arg);
```

Here `arg` is an alias for `&p2.env`, but `arg` has the *opened type*, in this case β`*`.

[4]Reference patterns also allow mutating fields of discriminated-union variants, which is why we originally added them to Cyclone.

At this point, we have created existential packages, used assignment to modify memory that has an existential type, and used reference patterns to get aliases of fields. It appears that we have a smooth integration of several features that are natural for a language at the C level of abstraction. Unfortunately, these features conspire to violate type safety:

```
void f(int* ptr) {
  struct T p1 = T(ignore, 0xABC);
  struct T p2 = T(assign, ptr);
  let T{<β> .f=g, .env=*arg} = p2;
  p2 = p1;
  g(37,*arg);
}
```

The call `g(37,*arg)` executes `assign` with `37` and `0xABC`—we are passing an `int` where we expect an `int*`, allowing us to write to an arbitrary address.

What went wrong in the type system? We used β to express an equality between one of `g`'s parameter types and the type of value at which `arg` points. But after the assignment, which changes `p2`'s witness type, this equality is false.

We have developed two solutions. The first solution forbids using reference patterns to match against fields of existential packages. Other uses of reference patterns are sound because assignment to a package mutates only the fields of the package. We call this solution, "no aliases at the opened type." The second solution forbids assigning to an existential package (or an aggregate value that has an existential package as a field). We call this solution, "no witness changes."

These solutions are *independent*: Either suffices and we could use different solutions for different existential packages. That is, for each existential-type declaration we could let the programmer decide which restriction the compiler enforces. The current implementation supports only "no aliases at the opened type" because we believe it is more useful, but both solutions are easy to enforce.

To emphasize the exact source of the problem, we mention some aspects that are not problematic. First, pointers to witness types are not a problem. For example, given `struct T2 {<α> void f(int, α); α* env;};` and the pattern `T2{<β> .f=g,.env=arg}`, an intervening assignment changes a package's witness type but does *not* change the type of the value at which `arg` points. Second, assignment to a *pointer to* an existential package is not a problem because it changes which package a pointer refers to, but does *not* change any package's witness type. Third, it is well-known that the typing rule for opening an existential package must forbid the introduced type variable from occurring in the type assigned to the term in which the type variable is in scope. In our case, this term is a statement, which has no type (or a unit type if you prefer), so this condition is trivially satisfied.

Multithreading introduces a similar problem that Chapter 5 addresses: The existential unpack is unsound if the witness can change in-between the binding of `g` and `arg`. We must exclude a witness change while binding a package's fields.

3.3.3 Informal Comparison of Problems

The potential problems discussed above both result from quantified types, aliasing, and mutation, so it is natural to suppose they are logical duals of the same problem. I have not found the correspondence between the two issues particularly illuminating, but I nonetheless point out similarities that may suggest a duality. Related work on polymorphic references is discussed in more detail in Section 3.6.

The polymorphic-reference example assigns to a variable at an *instantiated* type and *then* instantiates the same variable at a different type. In contrast, the existential-package example assigns to a value at an *unopened* type only *after* creating an alias at the opened type.

The ML "value restriction" is a very clever way to prevent types like $\forall\alpha.(\alpha*)$ by exploiting that expressions of such types cannot be values in ML. It effectively prevents certain types for a mutable locations' *contents*. In contrast, the "no witness changes" solution prevents certain types for a mutation's *location*.

With the exception of linear type systems, I know of no treatment of universal types that actually permits the types of values at mutable locations to change, as the "no aliases at the opened type" solution does. It is unclear what an invariant along these lines would look like for polymorphic references.

3.4 Evaluation

To evaluate the Cyclone features presented in this chapter qualitatively, we start with an optimistic assessment of what the features provide. We then describe some disappointing limitations and how future work might address them.

3.4.1 Good News

Type variables provide compile-time equalities of unknown types. Compared to C, they describe interfaces for polymorphic code and abstract types more precisely than `void*`. Compared to safe languages without them, they provide more code reuse.

Existential types give programmers first-class abstract data types without sacrificing C-like control over data representation. Building and using existential

packages does not "look" much like C code, but the difference is local. Put another way, porting C code that used a `struct` that converts easily to an existential type would require changing only function bodies that access fields of the `struct`. No restructuring of the code should be necessary. However, the existential types in this chapter "hide too much"—Chapters 4 and 5 will modify them to "leak" more information.

Type constructors provide an elegant way to describe container types (lists, dictionaries, hashtables, etc.) and universal quantification describes polymorphic routines over the types. The Cyclone implementation includes a powerful collection of container-type libraries that applications have used extensively. Using the libraries requires no more notation or overhead than in C, but we gain the advantage that we cannot use `void*` to confuse types.

In general, default annotations and intraprocedural type inference allow programmers to write little more than what is necessary for type safety. Writing, "`void swap(`α`*, `α`*)`" does not feel burdensome, and there could hardly exist a more concise way to express the important type equality.

Type constructors and abstract types also allow clever programmers to use the type system to encode restrictions on how clients can use a library. One fairly well-known trick is to use so-called *phantom types* [79] (type variables that are not used in the type's implementation), as in this example interface:

```
struct Read;
struct Write;
struct MyFile<α>;
struct MyFile<struct Read*>*  open_read(char*);
struct MyFile<struct Write*>* open_write(char*);
char read(struct MyFile<struct Read*>*);
void write(struct MyFile<struct Write*>*, char);
void reset(struct MyFile<α>*);
void close(struct MyFile<α>*);
```

This interface prevents reading a `MyFile` that was opened for writing or writing a `MyFile` that was opened for reading. Yet polymorphism allows closing or resetting any `MyFile`. The implementation of `struct MyFile` does not need run-time information indicating "read" or "write." Phantom types have their limits, however. We cannot soundly provide a function that changes a `MyFile` from "read" to "write" because a client can keep an alias with the "old" type. Similarly, the interface does not require that clients call `close` only once for each `MyFile`.

Cyclone's kind distinction is no more burdensome than in C, where abstract types must occur under pointers and `struct` types cannot be converted to `void*`.

Some of the inconvenience is inherent to exposing data representation; it is infeasible to support polymorphism over types of different sizes and calling conventions without imposing run-time cost or duplicating code. Nonetheless, C provides little support for abstract types, so it is a bit too easy to accept being "as good as C." Section 3.4.2 explores some possible improvements.

Restricting where programmers can introduce type quantifiers (universal quantification only on function types and existential quantification only on `struct` types) is usually not too restrictive. To see why, consider this small formal grammar for types:

$$\tau ::= \alpha \mid \text{int} \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \tau* \mid \exists\alpha.\tau \mid \forall\alpha.\tau$$

Types can be type variables, int, function types, pair types (i.e., anonymous `struct` types), pointer types, existential types, or universal types. Unlike the Cyclone implementation, this grammar does not restrict the form of quantified types. We argue informally why the generality is not very useful:

- $\forall\alpha.\alpha$ should not describe any value; nothing should have every type. $\exists\alpha.\alpha$ could describe any value (ignoring kind distinctions), but expressions of this type are unusable. For $\forall\alpha.\beta$ and $\exists\alpha.\beta$, we can just use β.
- For $\forall\alpha$.int and $\exists\alpha$.int, we can just use int.
- Cyclone provides $\forall\alpha.\tau_1 \rightarrow \tau_2$. For $\exists\alpha.\tau_1 \rightarrow \tau_2$, if α appears in τ_1, expressions of this type are unusable because we cannot call the function. Otherwise, we can just use $\tau_1 \rightarrow \exists\alpha.\tau_2$.
- Cyclone provides the analogue of $\exists\alpha.\tau_1 \times \tau_2$. For $\forall\alpha.\tau_1 \times \tau_2$, a similar value of type $(\forall\alpha.\tau_1) \times (\forall\alpha.\tau_2)$ is strictly more useful. Constructing such a similar value is easy because type-checking expressions of type τ_1 (respectively τ_2) does not exploit the type of τ_2 (respectively τ_1).
- For $\forall\alpha.(\tau*)$ and $\exists\alpha.(\tau*)$, we can just use $(\forall\alpha.\tau)*$ and $(\exists\alpha.\tau)*$, respectively. Note that $\forall\alpha.(\tau*)$ should not describe mutable values.

However, it would be useful to allow $\forall\alpha.\tau_1 + \tau_2$, where $\tau_1 + \tau_2$ is a (disjoint) *sum type*, especially in conjunction with abstract types. We return to Cyclone notation for an example. Suppose we want to implement an abstract list library. We can write the following (recalling that τ`@` describes pointers that cannot be `NULL` and `new` allocates new memory):

```
struct L1<α> {
  α hd;
  struct L1<α> *tl;
};
struct L<α> { struct L1<α> * x; };
struct L<α>@ empty() { return new L{.x=NULL}; }
struct L<α>@ cons(α x, struct L<α>@ l) { ... }
...
```

We can keep the implementation abstract from clients by declaring just `struct L<α>;`. Lists (`struct L<α>@`) are really a sum type because the `x` field is either `NULL` or a pointer. Because all empty lists have the same representation—regardless of the element type—it wastes space to allocate memory on each call to `empty`. To avoid this waste, we need something like:

```
(∀α struct L<α>) mt = L{.x=NULL};
struct L<α>@ empty() { return &(mt<α>); }
```

Of course, the type of the variable `mt` uses universal quantification. It also suffers from the polymorphic-reference problem (note that a type instantiation appears in a left-side expression), so we need to prohibit mutation for all types constructed from `struct L`. Without these additions, clients can share empty lists for each element type, but they cannot share empty lists for different element types.

3.4.2 Bad News

Cyclone provides no compile-time refinement of abstract types. As a simple example, it is tempting to allow programs like this one:

```
void swap(α*, α*);
void f(α* x, β* y) {
  if(*x == *y)
    swap(x,y);
}
```

The idea assumes that if two values are the same, then their types are the same. In the true-branch of the if-statement, the type-checker could include the *constraint* $\alpha = \beta$. Although this addition has questionable utility, it is tempting because we have constraints describing equalities and inequalities for the type-level variables we introduce in subsequent chapters. In particular, in Chapter 7 we use term-level tests to introduce type-level constraints. Because a primary goal of this dissertation

is to demonstrate that the same tools are useful and meaningful for a variety of problems, constraints for type variables merit consideration.

Unfortunately, the refinement in the above example is unsound because the assumption underlying it does not hold. Suppose α is `int` and β is `int*`. The condition `*x == *y` might still hold, allowing `swap` to put an `int` where we expect a pointer.

Subtyping also makes it unsound to use pointer-equality checks to introduce equality constraints. If α and β obey a strict subtype relationship, values of the types could be equal, but it is unsound to introduce a type-equality constraint.

However, there are sound ways for term-level tests (other than pointer equality) to introduce type-level constraints. For example, the type system in Chapter 7 can express roughly, "if some constant integer is 0, then α is τ." However, that system works only for constant integers; safe C programs may check more complex properties to determine a value's type. A more principled approach provides explicit "representation-type" terms for describing the types of other terms [215]. Because these terms are separate from the terms of the types they describe, they should work well in a language that exposes data representation. They are important for writing certain generic functions like marshallers and garbage collectors.

An easier and more justifiable addition is explicit subtyping constraints of the form $\tau_1 < \tau_2$. Adding such constraints as preconditions for polymorphic functions achieves *bounded quantification*, as in this example:

```
α f(void g(τ), α x : α<τ) {
  g(x);
  return x;
}
```

The constraint $\alpha<\tau$ requires any instantiation of `f`'s type to use a subtype of τ. In the body of `f`, we can soundly assume the constraint holds, so we use subsumption to type-check the function call. Without bounded quantification, the most permissive type for `f` would give `x` and the result the type τ. But then callers of `f` using a strict subtype of τ could not assume that the result of the call had the subtype. Section 3.6 discusses some known problems with bounded quantification.

As for the kind system, the information about type variables of kind `A` is extremely coarse, just like abstract `struct` types in C. One cannot, for example, write a function that works for all arrays with elements that are eight bytes long. Adding more descriptive kinds is straightforward. For example, `A8` could describe all types τ such that `sizeof(`τ`)==8` so long as the types have the same alignment constraints, calling convention, etc. Subkinding would make `A8` a subkind of `A`. However, `sizeof(`τ`)` is implementation-dependent, so portable code cannot assume its value. We could consider refining kind information within the program.

For example, for `if(sizeof(`τ`)==8)` s, we could give τ kind `A8` in s. Typed assembly languages can have such kinds because all sizes are known [155, 51].

I believe a better solution is to recognize that C-level tasks inherently include nonportable parts that can still benefit from language support. Most of an application should not make implementation-dependent assumptions, and the language implementation should check this property automatically. But when real bit-level data representation and calling convention matter, an application should be able to specify its assumptions about the implementation and have the compiler check the code accordingly. In terms of our example, the code for manipulating arrays with 8-byte elements remains portable, but an implementation-dependent assumption guards the use of it for type τ. Similar assumptions could allow other reasonable operations, such as casting between `struct T1 { int x; };` and `struct T2 { char y[4]; };` on appropriate architectures. Instead, Cyclone is like C, a strange hybrid that exposes data representation in terms of field order, levels of indirection, etc., but without committing to the size of types or the alignment of fields.

As mentioned previously, we could relax the rules about where abstract types appear by duplicating code for every type at which it is instantiated. This approach is closer to C++ templates [193]. It is a valuable alternative for widely used, performance-critical libraries, such as hashtables, where a level of indirection can prove costly. However, it is difficult to maintain separate compilation. Polymorphic recursion is also a problem because it takes care to bound the amount of generated code. For example, this program would need an infinite amount of code.

```
struct T<α:A> {α x; α y; }; // not legal Cyclone
void f<α:A>(struct T<α> t) {
  struct T<struct T<α>> bigger = T{.x=t, .y=t};
  f(bigger);
}
```

We have avoided this design path in Cyclone, largely because the C++ designers have explored it extensively.

The inability of the Cyclone type system to express restrictions on aliases to locations causes Cyclone to forbid some safe programs. For example, given a pointer to an α, it is safe to store a β at the pointed-to location "temporarily," *provided that no code expecting an α reads the location before it again holds an α*. If no aliases to the location exist, this property is much easier to check statically. As another example, we can allow reference patterns for fields of mutable existential packages, provided no (witness-changing) mutation occurs before the variable bound with the reference pattern is dereferenced. Restricted aliasing makes it possible to check that no such mutation occurs.

Finally, some small problems in Cyclone's design of type variables and casts deserve brief mention. First, like in C, a cast's meaning is type-dependent. For example, casting a `float` to an `int` does not treat the same bit-sequence as an integer. A cleaner design would distinguish coercive casts (which have run-time effect) from other ones. Similar distinctions exist in C++.

Second, forbidding direct access to existential-package fields is inconvenient. Perhaps a simple flow analysis could infer the unpacking implicit in field access without violating soundness.

Third, partial instantiation of type constructors and polymorphic functions is sometimes less convenient than I have suggested. The instantiation is in order, which means the type-constructor and function creator determines what partial applicatons are allowed. (The same shortcoming exists at the term level in functional languages with currying.) Moreover, the partial applications described in this chapter are just shorthand for implicit full applications. But sometimes it is necessary to partially instantiate a universal type and delay the rest of the instantiation. (I have extended the Cyclone implementation to support such a true partial instantiation. The example where I found it necessary involves memory management; see Chapter 4.)

Fourth, Cyclone does not have *higher-order* type constructors. There is no way to parameterize one type constructor by another type constructor. To date, there has not been sufficient demand to implement this feature.

3.5 Formalism

To investigate the soundness of the features presented in this chapter, especially in the presence of the complications described in Sections 3.2 and Section 3.3, we develop a formal abstract machine and a type system for it. This machine defines programs that manipulate a heap of mutable locations. Locations can hold integers or pointers. The machine gets "stuck" if a program tries to dereference an integer. The type system has universal quantification and existential quantification (with both solutions from Section 3.3). The theorem in Section 3.5.4 ensures well-typed programs never lead to stuck machines.

As usual, a formal model lets us give precise meaning to our language-design ideas, ignore issues orthogonal to safety (e.g., concrete syntax and floating-point numbers), and prove a rigorous result. To keep the model and proof tractable, we make further simplifications, such as omitting type constructors and memory management. An inherent trade-off exists between simplifying to focus on relevant issues and potentially missing an actual unsoundness due to a subtle interaction.

Section 3.5.1 defines the syntax of programs and program states. Section 3.5.2

presents the rules for how the machine executes. Section 3.5.3 presents the type system. In practice, we use the static semantics only for source programs, but the type-safety proof requires extending the type system to type-check program states. Before proceeding, we emphasize the most novel aspects of our formalism:

1. Like Cyclone and C, we distinguish left-expressions, right-expressions, and statements. The definitions for these classes of terms are mutually inductive, so the dynamic and static semantics comprise interdependent judgments.

2. Functions must execute return statements. (Our formalism does not have `void`; Cyclone does.) A separate judgment encodes a simple syntax-directed analysis to ensure a function cannot terminate without returning. (The actual Cyclone implementation uses a flow analysis.)

3. We allow aliasing of mutable fields (e.g., `&x`$.i.j$) and assignment to aggregate values (e.g., `x`$.i$`=`e where `x`$.i$ is itself an aggregate). This feature complicates the rules for accessing, mutating, and type-checking aggregates.

4. We classify types with kinds `B` and `A`. The type system prohibits programs that would need to know the size of a type variable of kind `A`.

5. To support both our solutions for mutable existential packages, the syntax distinguishes two styles of existential types. The type system defines the set of "assignable" types to disallow some witness changes. Moreover, the type-safety proof requires the type system to maintain the witness types for packages used in reference patterns. Otherwise, the induction hypothesis would not be strong enough to show that evaluation preserves typing.

The formalisms in subsequent chapters also include the first two features, so we describe them in some detail in this chapter's simpler setting. Without them, the abstract machine would "look" much less like C. The third feature also models an important part of C. However, it is cumbersome, so after Chapter 4, we further restrict left-expressions to prevent taking the address of fields. This later restriction is only for simplicity. The last two features capture this chapter's most interesting aspects. Subsequent formalisms avoid the complications these features introduce by disallowing type variables of kind `A` and eliminating reference patterns. Such expediency in this chapter would be too simple.

3.5.1 Syntax

Figure 3.1 presents the language's syntax. We model execution with a program state consisting of a heap (for the data) and a statement (for the control). For

the heap, we reuse variables to represent addresses, so the heap maps variables to values. We write $\cdot$ for the empty heap. We allow implicit reordering of heaps, so they act as partial maps.

Terms include expressions and statements. Statements include expressions (e) executed for effect, return statements ($\mathsf{return}\ e$), sequential composition ($s; s$), conditionals ($\mathsf{if}\ e\ s\ s$), and loops ($\mathsf{while}\ e\ s$). A variable binding ($\mathsf{let}\ x = e;\ s$) extends the heap with a binding for x, which we can assume is unique because the binding is α-convertible. Because memory management is not our present concern, the dynamic semantics never contracts the heap. There are two forms for destructing existential packages. The form $\mathsf{open}\ e\ \mathsf{as}\ \alpha, x;\ s$ binds x to a *copy* of the contents of the evaluation of e, whereas $\mathsf{open}\ e\ \mathsf{as}\ \alpha, *x;\ s$ binds x to a *pointer* to the contents of the evaluation of e. The latter form corresponds to reference patterns. For simplicity, it produces a pointer to the entire contents, not a particular field.

Expressions include integers (i); function definitions ($(\tau\ x) \to \tau\ e$) with explicit type parameters ($\Lambda\alpha{:}\kappa.f$); pointer creations ($\&e$); pointer dereferences ($*e$); pairs ((e_1, e_2)); field accesses ($e.i$); assignments ($e_1{=}e_2$); function calls ($e_1(e_2)$); type instantiations ($e[\tau]$); and existential packages ($\mathsf{pack}\ \tau', e\ \mathsf{as}\ \exists^{\phi}\alpha{:}\kappa.\tau$). In this package creation, τ' is the witness type. Its explicit mention is a technical convenience.

Two stranger expression forms remain. The call form ($\mathsf{call}\ s$) maintains the call stack in the term syntax: A function call is rewritten with this form and the function's return eliminates it. Instead of variables (x), we write variables with *paths* (p), so the expression form is xp. If p is the empty path ($\cdot$), then xp is like a variable x, and we often write x as short-hand for $x\cdot$. There is no need for nonempty paths in source programs. Because values may be pairs or packages, we use paths to refer to parts of values. A path is just a sequence of 0, 1, and `u`. As defined in the next section, 0 and 1 refer to pair components and `u` refers to the value inside an existential package. We write p_1p_2 for the sequence that is p_1 followed by p_2. We blur the distinction between sequences and sequence elements as convenient. So $0p$ means the path beginning with 0 and continuing with p and $p0$ means the path ending with 0 after p.

The valid left-expressions are a subset of the valid right-expressions. The type system enforces the restriction. Invalid left-expressions do not type-check when they occur under the & operator or on the left side of an assignment.

Types include type variables (α), a base type (int), products ($\tau_1 \times \tau_2$), pointers ($\tau*$), existentials ($\exists^{\phi}\alpha{:}\kappa.\tau$), and universals ($\forall\alpha{:}\kappa.\tau$). We consider quantified types equal up to systematic renaming of the bound type variable (α-conversion). Compared to Cyclone, we have replaced `struct` types with "anonymous" product types (pairs) and eliminated user-defined type constructors. Type-variable bindings include an explicit kind, κ. Because aliasing is relevant, all uses of pointers

$$
\begin{array}{rrcl}
\text{kinds} & \kappa & ::= & \mathtt{B} \mid \mathtt{A} \\
\text{types} & \tau & ::= & \alpha \mid \text{int} \mid \tau \times \tau \mid \tau \to \tau \mid \tau* \mid \forall \alpha{:}\kappa.\tau \mid \exists^{\phi} \alpha{:}\kappa.\tau \\
 & \phi & ::= & \delta \mid \& \\
\text{terms} & s & ::= & e \mid \mathsf{return}\ e \mid s; s \mid \mathsf{if}\ e\ s\ s \mid \mathsf{while}\ e\ s \mid \mathsf{let}\ x = e;\ s \\
 & & \mid & \mathsf{open}\ e\ \mathsf{as}\ \alpha, x;\ s \mid \mathsf{open}\ e\ \mathsf{as}\ \alpha, *x;\ s \\
 & e & ::= & xp \mid i \mid f \mid \&e \mid *e \mid (e, e) \mid e.i \mid e{=}e \mid e(e) \mid \mathsf{call}\ s \\
 & & \mid & e[\tau] \mid \mathsf{pack}\ \tau, e\ \mathsf{as}\ \tau \\
 & f & ::= & (\tau\ x) \to \tau\ s \mid \Lambda \alpha{:}\kappa.f \\
 & p & ::= & \cdot \mid ip \mid \mathtt{u}p \\
\text{values} & v & ::= & i \mid \&xp \mid f \mid (v, v) \mid \mathsf{pack}\ \tau, v\ \mathsf{as}\ \tau \\
\text{heaps} & H & ::= & \cdot \mid H, x \mapsto v \\
\text{states} & P & ::= & H; s \\
\text{contexts} & \Delta & ::= & \cdot \mid \alpha{:}\kappa \\
 & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
 & \Upsilon & ::= & \cdot \mid \Upsilon, xp{:}\tau \\
 & C & ::= & \Delta; \Upsilon; \Gamma
\end{array}
$$

Figure 3.1: Chapter 3 Formal Syntax

are explicit. In particular, a value of a product type is a record, not a pointer to a record. To distinguish our two approaches to existential types, we annotate $\exists$ with δ (allowing witness changes) or & (allowing aliases at the opened type).

As technical points, we treat the parts of a typing context (Δ, Γ, and Υ) as implicitly reorderable (and as partial maps) where convenient. When we write $\Gamma, x{:}\tau$, we assume $x \notin \text{Dom}(\Gamma)$. We write $\Gamma\Gamma'$ (and similarly for Δ and Υ) for the union of two contexts with disjoint domains, implicitly assuming disjointedness.

3.5.2 Dynamic Semantics

Six deterministic relations define the (small-step, operational) dynamic semantics. A program state $H; s$ becomes $H'; s'$ if the rules in Figure 3.2 establish $H; s \xrightarrow{\text{s}} H'; s'$. This relation and the related relations for expressions ($H; e \xrightarrow{\text{r}} H'; e'$ and $H; e \xrightarrow{\text{l}} H'; e'$ in Figure 3.3) are interdependent because statements and expressions can contain each other. The relations in Figure 3.4 describe how paths direct the access and mutation of values. Type substitution (Figure 3.5) gives operational meaning to $e[\tau]$ and open. Types play no essential run-time role, so we can view substitution as an effectless operation useful for proving type preservation. We now describe the six definitions in more detail.

$$\frac{x \notin \mathrm{Dom}(H)}{H; \mathsf{let}\ x = v;\ s \xrightarrow{\mathrm{s}} H, x \mapsto v; s}\ \text{DS3.1}$$

$$\frac{}{H; (v; s) \xrightarrow{\mathrm{s}} H; s}\ \text{DS3.2} \qquad \frac{}{H; (\mathsf{return}\ v; s) \xrightarrow{\mathrm{s}} H; \mathsf{return}\ v}\ \text{DS3.3}$$

$$\frac{}{H; \mathsf{if}\ 0\ s_1\ s_2 \xrightarrow{\mathrm{s}} H; s_2}\ \text{DS3.4} \qquad \frac{i \neq 0}{H; \mathsf{if}\ i\ s_1\ s_2 \xrightarrow{\mathrm{s}} H; s_1}\ \text{DS3.5}$$

$$\frac{}{H; \mathsf{while}\ e\ s \xrightarrow{\mathrm{s}} H; \mathsf{if}\ e\ (s; \mathsf{while}\ e\ s)\ 0}\ \text{DS3.6}$$

$$\frac{}{H; \mathsf{open}\ (\mathsf{pack}\ \tau', v\ \mathsf{as}\ \exists^{\phi}\alpha{:}\kappa.\tau)\ \mathsf{as}\ \alpha, x;\ s \xrightarrow{\mathrm{s}} H; \mathsf{let}\ x = v;\ s[\tau'/\alpha]}\ \text{DS3.7}$$

$$\frac{\mathrm{get}(H(x), p, \mathsf{pack}\ \tau', v\ \mathsf{as}\ \exists^{\phi}\alpha{:}\kappa.\tau)}{H; \mathsf{open}\ xp\ \mathsf{as}\ \alpha, *x';\ s \xrightarrow{\mathrm{s}} H; \mathsf{let}\ x' = \&xp\mathtt{u};\ s[\tau'/\alpha]}\ \text{DS3.8}$$

$$\frac{H; e \xrightarrow{\mathrm{r}} H'; e'}{\begin{array}{r} H; e \xrightarrow{\mathrm{s}} H'; e' \\ H; \mathsf{return}\ e \xrightarrow{\mathrm{s}} H'; \mathsf{return}\ e' \\ H; \mathsf{if}\ e\ s_1\ s_2 \xrightarrow{\mathrm{s}} H'; \mathsf{if}\ e'\ s_1\ s_2 \\ H; \mathsf{let}\ x = e;\ s \xrightarrow{\mathrm{s}} H'; \mathsf{let}\ x = e';\ s \\ H; \mathsf{open}\ e\ \mathsf{as}\ \alpha, x;\ s \xrightarrow{\mathrm{s}} H'; \mathsf{open}\ e'\ \mathsf{as}\ \alpha, x;\ s \end{array}}\ \text{DS3.9} \qquad \frac{H; s \xrightarrow{\mathrm{s}} H'; s'}{H; s; s_2 \xrightarrow{\mathrm{s}} H'; s'; s_2}\ \text{DS3.10}$$

$$\frac{H; e \xrightarrow{\mathrm{l}} H'; e'}{H; \mathsf{open}\ e\ \mathsf{as}\ \alpha, *x;\ s \xrightarrow{\mathrm{s}} H'; \mathsf{open}\ e'\ \mathsf{as}\ \alpha, *x;\ s}\ \text{DS3.11}$$

Figure 3.2: Chapter 3 Dynamic Semantics, Statements

Rule DS3.1 is the only rule that extends the heap. Because $\mathsf{let}\ x = v;\ s$ is α-convertible, we can assume x does not already name a heap location. Bindings exist forever, so a statement like $\mathsf{let}\ x = v;\ \mathsf{return}\ \&x$ is reasonable. Rules DS3.2–6 are unsurprising rules for simplifying sequences, conditionals, and loops. Rule DS3.7 uses a let to simplify the results of opening an existential package. In the result, α is not in scope, so we substitute the package's witness type for α in s. Rule DS3.8 also uses let, but it binds the variable to the *address of* the package's contents. To keep type-checking syntax-directed, we append $\mathtt{u}$ to the path. That way, we refer to the package's contents, not the package. The get relation, described below, is used here only to acquire the witness type we need for substitution. Rules DS3.9–11 are congruence rules, which evaluate terms contained in larger terms. Putting multiple conclusions in one rule is just for conciseness. The interesting distinction is that in

$$\frac{\mathrm{get}(H(x), p, v)}{H; xp \xrightarrow{\mathrm{r}} H; v}\ \text{DR3.1} \qquad \frac{\mathrm{set}(v', p, v, v'')}{H, x \mapsto v', H'; xp{=}v \xrightarrow{\mathrm{r}} H, x \mapsto v'', H'; v}\ \text{DR3.2}$$

$$\frac{}{H; *\&xp \xrightarrow{\mathrm{r}} H; xp}\ \text{DR3.3} \qquad \frac{}{H; (v_0, v_1).i \xrightarrow{\mathrm{r}} H; v_i}\ \text{DR3.4}$$

$$\frac{}{H; ((\tau\ x) \to \tau'\ s)(v) \xrightarrow{\mathrm{r}} H; \mathsf{call}\ (\mathsf{let}\ x = v;\ s)}\ \text{DR3.5}$$

$$\frac{}{H; \mathsf{call\ return}\ v \xrightarrow{\mathrm{r}} H; v}\ \text{DR3.6} \qquad \frac{}{H; (\Lambda\alpha{:}\kappa.f)[\tau] \xrightarrow{\mathrm{r}} H; f[\tau/\alpha]}\ \text{DR3.7}$$

$$\frac{H; s \xrightarrow{\mathrm{s}} H'; s'}{H; \mathsf{call}\ s \xrightarrow{\mathrm{r}} H'; \mathsf{call}\ s'}\ \text{DR3.8} \qquad \frac{H; e \xrightarrow{\mathrm{l}} H'; e'}{\begin{array}{c} H; \&e \xrightarrow{\mathrm{r}} H'; \&e' \\ H; e{=}e_2 \xrightarrow{\mathrm{r}} H'; e'{=}e_2 \end{array}}\ \text{DR3.9}$$

$$\frac{H; e \xrightarrow{\mathrm{r}} H'; e'}{\begin{array}{cc} H; *e \xrightarrow{\mathrm{r}} H'; *e' & H; (e, e_2) \xrightarrow{\mathrm{r}} H'; (e', e_2) \\ H; e.i \xrightarrow{\mathrm{r}} H'; e'.i & H; (v, e) \xrightarrow{\mathrm{r}} H'; (v, e') \\ H; xp{=}e \xrightarrow{\mathrm{r}} H'; xp{=}e' & H; e(e_2) \xrightarrow{\mathrm{r}} H'; e'(e_2) \\ H; e[\tau] \xrightarrow{\mathrm{r}} H'; e'[\tau] & H; v(e) \xrightarrow{\mathrm{r}} H'; v(e') \\ \multicolumn{2}{c}{H; \mathsf{pack}\ \tau', e\ \mathsf{as}\ \exists^{\phi}\alpha{:}\kappa.\tau \xrightarrow{\mathrm{r}} H'; \mathsf{pack}\ \tau', e'\ \mathsf{as}\ \exists^{\phi}\alpha{:}\kappa.\tau} \end{array}}\ \text{DR3.10}$$

$$\frac{}{H; (xp).i \xrightarrow{\mathrm{l}} H; xpi}\ \text{DL3.1} \qquad \frac{}{H; *\&xp \xrightarrow{\mathrm{l}} H; xp}\ \text{DL3.2}$$

$$\frac{H; e \xrightarrow{\mathrm{r}} H'; e'}{H; *e \xrightarrow{\mathrm{l}} H'; *e'}\ \text{DL3.3} \qquad \frac{H; e \xrightarrow{\mathrm{l}} H'; e'}{H; e.i \xrightarrow{\mathrm{l}} H'; e'.i}\ \text{DL3.4}$$

Figure 3.3: Chapter 3 Dynamic Semantics, Expressions

$$\frac{}{\mathrm{get}(v, \cdot, v)} \qquad \frac{\mathrm{get}(v_0, p, v)}{\mathrm{get}((v_0, v_1), 0p, v)} \qquad \frac{\mathrm{get}(v_1, p, v)}{\mathrm{get}((v_0, v_1), 1p, v)}$$

$$\frac{\mathrm{get}(v_1, p, v)}{\mathrm{get}(\mathsf{pack}\ \tau', v_1\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau,\ \mathtt{u}p,\ v)}$$

$$\frac{}{\mathrm{set}(v', \cdot, v, v)} \qquad \frac{\mathrm{set}(v_0, p, v, v')}{\mathrm{set}((v_0, v_1), 0p, v, (v', v_1))} \qquad \frac{\mathrm{set}(v_1, p, v, v')}{\mathrm{set}((v_0, v_1), 1p, v, (v_0, v'))}$$

$$\frac{\mathrm{set}(v_1, p, v, v')}{\mathrm{set}(\mathsf{pack}\ \tau', v_1\ \mathsf{as}\ \exists^{\phi}\alpha{:}\kappa.\tau,\ \mathtt{u}p,\ v,\ \mathsf{pack}\ \tau', v'\ \mathsf{as}\ \exists^{\phi}\alpha{:}\kappa.\tau)}$$

Figure 3.4: Chapter 3 Dynamic Semantics, Heap Objects

$\mathsf{open}\ e\ \mathsf{as}\ \alpha, x;\ s$, the expression e is a right-expression, but in $\mathsf{open}\ e\ \mathsf{as}\ \alpha, *x;\ s$, it is a left-expression.

Right-expressions evaluate to values using rules DR3.1–10. The get and set relations handle the details of reading and mutating heap locations (DR3.1 and DR3.2). Rules DR3.3 and DR3.4 eliminate pointers and pairs, respectively. Rules DR3.5 and DR3.6 introduce and eliminate function calls, using a let to pass the function argument. Rule DR3.7 uses type substitution for instantiation. Rules DR3.8–10 are the congruence rules. Note that the evaluation order is left-to-right and that DR3.9 indicates the left-expression positions.

Left-expressions evaluate to something of the form xp. We need few rules because the type system restricts the form of left-expressions. The only interesting rule is DL3.1, which appends a field projection to the path. To contrast left-expressions and right-expressions, compare the results of DL3.2 and DR3.3. For left-expressions, the result is a terminal form (no rule applies), but for right-expressions, rule DR3.1 applies.

The get relation defines the use of paths to destruct values. As examples, $\mathrm{get}((v_0, v_1), 1, v_1)$ and $\mathrm{get}(\mathsf{pack}\ \tau', v\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau,\ \mathtt{u},\ v)$. That is, we use $\mathtt{u}$ to get a package's contents, which we never do if the witness might change. The set relation defines the use of paths to update parts of values: $\mathrm{set}(v_1, p, v_2, v_3)$ means updating the part of v_1 corresponding to p with v_2 produces v_3. For example, $\mathrm{set}((v_1, ((v_2, v_3), v_4)), 10, (v_5, v_6), (v_1, ((v_5, v_6), v_4)))$.

Type substitution is completely straightforward. We replace free occurrences of the type variable with the type. Subsequent chapters omit the uninteresting cases of the definition. In this chapter, no cases are interesting.

As an example of the dynamic semantics, here is a variation of the previous

$$
\begin{array}{lrcl}
\text{types:} & \alpha[\tau/\alpha] & = & \tau \\
& \beta[\tau/\alpha] & = & \beta \\
& \text{int}[\tau/\alpha] & = & \text{int} \\
& (\tau_0 \times \tau_1)[\tau/\alpha] & = & \tau_0[\tau/\alpha] \times \tau_1[\tau/\alpha] \\
& (\tau_1 \to \tau_2)[\tau/\alpha] & = & \tau_1[\tau/\alpha] \to \tau_2[\tau/\alpha] \\
& (\tau' *)[\tau/\alpha] & = & \tau'[\tau/\alpha] * \\
& (\forall \beta{:}\kappa.\tau')[\tau/\alpha] & = & \forall \beta{:}\kappa.\tau'[\tau/\alpha] \\
& (\exists^{\phi} \beta{:}\kappa.\tau')[\tau/\alpha] & = & \exists^{\phi} \beta{:}\kappa.\tau'[\tau/\alpha] \\
\text{contexts:} & \cdot[\tau/\alpha] & = & \cdot \\
& (\Gamma, x{:}\tau')[\tau/\alpha] & = & \Gamma[\tau/\alpha], x{:}\tau'[\tau/\alpha] \\
\text{expressions:} & xp[\tau/\alpha] & = & xp \\
& i[\tau/\alpha] & = & i \\
& (\&e)[\tau/\alpha] & = & \&(e[\tau/\alpha]) \\
& (*e)[\tau/\alpha] & = & *(e[\tau/\alpha]) \\
& (e_0, e_1)[\tau/\alpha] & = & (e_0[\tau/\alpha], e_1[\tau/\alpha]) \\
& (e.i)[\tau/\alpha] & = & (e[\tau/\alpha]).i \\
& (e_1{=}e_2)[\tau/\alpha] & = & (e_1[\tau/\alpha]{=}e_2[\tau/\alpha]) \\
& (e_1(e_2))[\tau/\alpha] & = & e_1[\tau/\alpha](e_2[\tau/\alpha]) \\
& (\mathsf{call}\ s)[\tau/\alpha] & = & \mathsf{call}\ (s[\tau/\alpha]) \\
& (e[\tau'])[\tau/\alpha] & = & (e[\tau/\alpha])[\tau'[\tau/\alpha]] \\
& (\mathsf{pack}\ \tau_1, e\ \mathsf{as}\ \exists^{\phi} \beta{:}\kappa.\tau_2)[\tau/\alpha] & = & \mathsf{pack}\ \tau_1[\tau/\alpha], e[\tau/\alpha]\ \mathsf{as}\ \exists^{\phi} \beta{:}\kappa.\tau_2[\tau/\alpha] \\
& ((\tau_1\ x) \to \tau_2\ s)[\tau/\alpha] & = & (\tau_1[\tau/\alpha]\ x) \to \tau_2[\tau/\alpha]\ s[\tau/\alpha] \\
& (\Lambda \beta{:}\kappa.f)[\tau/\alpha] & = & \Lambda \beta{:}\kappa.f[\tau/\alpha] \\
\text{statements:} & e[\tau/\alpha] & = & e[\tau/\alpha] \quad \text{(right side is an expression)} \\
& (\mathsf{return}\ e)[\tau/\alpha] & = & \mathsf{return}\ e[\tau/\alpha] \\
& (s_1; s_2)[\tau/\alpha] & = & s_1[\tau/\alpha]; s_2[\tau/\alpha] \\
& (\mathsf{while}\ e\ s)[\tau/\alpha] & = & \mathsf{while}\ e[\tau/\alpha]\ s[\tau/\alpha] \\
& (\mathsf{if}\ e\ s_1\ s_2)[\tau/\alpha] & = & \mathsf{if}\ e[\tau/\alpha]\ s_1[\tau/\alpha]\ s_2[\tau/\alpha] \\
& (\mathsf{let}\ x = e;\ s)[\tau/\alpha] & = & \mathsf{let}\ x = e[\tau/\alpha];\ s[\tau/\alpha] \\
& (\mathsf{open}\ e\ \mathsf{as}\ \beta, x;\ s)[\tau/\alpha] & = & \mathsf{open}\ e[\tau/\alpha]\ \mathsf{as}\ \beta, x;\ s[\tau/\alpha] \\
& (\mathsf{open}\ e\ \mathsf{as}\ \beta, *x;\ s)[\tau/\alpha] & = & \mathsf{open}\ e[\tau/\alpha]\ \mathsf{as}\ \beta, *x;\ s[\tau/\alpha]
\end{array}
$$

Note: Throughout, we mean $\beta \neq \alpha$ and implicitly rename to avoid capture.

Figure 3.5: Chapter 3 Dynamic Semantics, Type Substitution

unsoundness example. We use assignment instead of function pointers, but the idea is the same. For now, we do not specify the style of the existential types.

(1) let $x_{zero} = 0$;
(2) let $x_{pzero} = \&x_{zero}$;
(3) let $x_{pkg} =$ pack int$*, (\&x_{pzero}, x_{pzero})$ as $\exists^{\phi}\alpha$:B.$\alpha* \times \alpha$;
(4) open x_{pkg} as $\beta, *x_{pr}$;
(5) let $x_{fst} = (*x_{pr}).0$;
(6) $x_{pkg}=$pack int$, (x_{pzero}, x_{zero})$ as $\exists^{\phi}\alpha$:B.$\alpha* \times \alpha$;
(7) $*x_{fst}=(*x_{pr}).1$;
(8) $*x_{pzero}=x_{zero}$

Lines (1)–(5) allocate values in the heap. After line (3), location x_{pkg} contains pack int$*, (\&x_{pzero}\cdot, \&x_{zero}\cdot)$ as $\exists^{\phi}\alpha$:B.$\alpha* \times \alpha$. Line (4) substitutes int$*$ for β and location x_{pr} contains $\&x_{pkg}$u. After line (6), x_{fst} contains $\&x_{pzero}\cdot$ and x_{pkg} contains pack int$, (\&x_{zero}, 0)$ as $\exists^{\phi}\alpha$:B.$\alpha* \times \alpha$. Hence line (7) assigns 0 to x_{pzero}, which causes line (8) to be stuck because there is no H, H', and e' for which $H; *0 \xrightarrow{1} H'; e'$.

To complete the example, we need to choose δ or & for each ϕ. Fortunately, as the next section explains, no choice produces a well-typed program.

The type information associated with packages and paths keeps type-checking syntax-directed. We could define an erasure function over heaps that replaces pack τ', v as $\exists^{\phi}\alpha{:}\kappa.\tau$ with v and removes u from paths. It should be straightforward to prove that erasure and evaluation commute (for a semantics that treats open like let).

3.5.3 Static Semantics

Because program execution begins with an empty heap, a source program is just a statement s. To allow s, we require $\cdot;\cdot;\cdot;\tau \vdash_{\text{styp}} s$ (for some type τ) and $\vdash_{\text{ret}} s$, using the rules in Figures 3.7 and 3.10, respectively. The former ensures conventional type-checking; terms are never used with inappropriate operations and never refer to undefined variables. The latter ensures that s does not terminate without executing a return statement.

The $\vdash_{\text{styp}}$ judgment and the type-checking judgments for right-expressions and left-expressions ($\vdash_{\text{rtyp}}$ and $\vdash_{\text{ltyp}}$ in Figure 3.8) are interdependent, just like the corresponding run-time relations. The strangest part of these judgments is Υ, which is irrelevant in source programs. As described below, it captures the invariant that packages used in terms of the form open e as $\alpha, *x;\ s$ are never mutated. The gettype relation (Figure 3.9) is the static analogue of the get relation. We use it to type-check paths.

$$\frac{}{\Delta \vdash_{\mathrm{k}} \mathrm{int} : \mathtt{B}} \qquad \frac{}{\Delta, \alpha{:}\mathtt{B} \vdash_{\mathrm{k}} \alpha : \mathtt{B}} \qquad \frac{}{\Delta, \alpha{:}\mathtt{A} \vdash_{\mathrm{k}} \alpha* : \mathtt{B}} \qquad \frac{\Delta \vdash_{\mathrm{k}} \tau : \mathtt{B}}{\Delta \vdash_{\mathrm{k}} \tau : \mathtt{A}}$$

$$\frac{\Delta \vdash_{\mathrm{k}} \tau_0 : \mathtt{A} \quad \Delta \vdash_{\mathrm{k}} \tau_1 : \mathtt{A}}{\begin{array}{c}\Delta \vdash_{\mathrm{k}} \tau_0 \times \tau_1 : \mathtt{A} \\ \Delta \vdash_{\mathrm{k}} \tau_0 \to \tau_1 : \mathtt{A}\end{array}} \qquad \frac{\Delta \vdash_{\mathrm{k}} \tau : \mathtt{A}}{\Delta \vdash_{\mathrm{k}} \tau* : \mathtt{B}} \qquad \frac{\Delta, \alpha{:}\kappa \vdash_{\mathrm{k}} \tau : \mathtt{A} \quad \alpha \notin \mathrm{Dom}(\Delta)}{\begin{array}{c}\Delta \vdash_{\mathrm{k}} \forall \alpha{:}\kappa.\tau : \mathtt{A} \\ \Delta \vdash_{\mathrm{k}} \exists^{\phi} \alpha{:}\kappa.\tau : \mathtt{A}\end{array}}$$

$$\frac{\Delta \vdash_{\mathrm{k}} \tau : \kappa}{\Delta \vdash_{\mathrm{ak}} \tau : \kappa} \qquad \frac{}{\Delta, \alpha{:}\mathtt{A} \vdash_{\mathrm{ak}} \alpha : \mathtt{A}}$$

$$\frac{}{\Delta \vdash_{\mathrm{asgn}} \mathrm{int}} \qquad \frac{}{\Delta, \alpha{:}\mathtt{B} \vdash_{\mathrm{asgn}} \alpha} \qquad \frac{}{\Delta \vdash_{\mathrm{asgn}} \tau*}$$

$$\frac{\Delta \vdash_{\mathrm{asgn}} \tau_0 \quad \Delta \vdash_{\mathrm{asgn}} \tau_1}{\begin{array}{c}\Delta \vdash_{\mathrm{asgn}} \tau_0 \times \tau_1 \\ \Delta \vdash_{\mathrm{asgn}} \tau_0 \to \tau_1\end{array}} \qquad \frac{\Delta, \alpha{:}\kappa \vdash_{\mathrm{asgn}} \tau}{\begin{array}{c}\Delta \vdash_{\mathrm{asgn}} \forall \alpha{:}\kappa.\tau \\ \Delta \vdash_{\mathrm{asgn}} \exists^{\delta} \alpha{:}\kappa.\tau\end{array}}$$

$$\frac{}{\Delta \vdash_{\mathrm{wf}} \cdot} \qquad \frac{\Delta \vdash_{\mathrm{wf}} \Gamma \quad \Delta \vdash_{\mathrm{k}} \tau : \mathtt{A}}{\Delta \vdash_{\mathrm{wf}} \Gamma, x{:}\tau} \qquad \frac{}{\vdash_{\mathrm{wf}} \cdot} \qquad \frac{\vdash_{\mathrm{wf}} \Upsilon \quad \cdot \vdash_{\mathrm{k}} \tau : \mathtt{A}}{\vdash_{\mathrm{wf}} \Upsilon, xp{:}\tau}$$

$$\frac{\Delta \vdash_{\mathrm{wf}} \Gamma \quad \vdash_{\mathrm{wf}} \Upsilon}{\vdash_{\mathrm{wf}} \Delta; \Upsilon; \Gamma}$$

Figure 3.6: Chapter 3 Kinding and Context Well-Formedness

$$\frac{C \vdash_{\mathrm{rtyp}} e : \tau'}{C; \tau \vdash_{\mathrm{styp}} e} \ \mathrm{SS3.1} \qquad \frac{C \vdash_{\mathrm{rtyp}} e : \tau}{C; \tau \vdash_{\mathrm{styp}} \mathsf{return}\ e} \ \mathrm{SS3.2} \qquad \frac{C; \tau \vdash_{\mathrm{styp}} s_1 \quad C; \tau \vdash_{\mathrm{styp}} s_2}{C; \tau \vdash_{\mathrm{styp}} s_1; s_2} \ \mathrm{SS3.3}$$

$$\frac{C \vdash_{\mathrm{rtyp}} e : \mathrm{int} \quad C; \tau \vdash_{\mathrm{styp}} s}{C; \tau \vdash_{\mathrm{styp}} \mathsf{while}\ e\ s} \ \mathrm{SS3.4} \qquad \frac{C \vdash_{\mathrm{rtyp}} e : \mathrm{int} \quad C; \tau \vdash_{\mathrm{styp}} s_1 \quad C; \tau \vdash_{\mathrm{styp}} s_2}{C; \tau \vdash_{\mathrm{styp}} \mathsf{if}\ e\ s_1\ s_2} \ \mathrm{SS3.5}$$

$$\frac{\Delta; \Upsilon; \Gamma, x{:}\tau'; \tau \vdash_{\mathrm{styp}} s \quad \Delta; \Upsilon; \Gamma \vdash_{\mathrm{rtyp}} e : \tau' \quad x \notin \mathrm{Dom}(\Gamma)}{\Delta; \Upsilon; \Gamma; \tau \vdash_{\mathrm{styp}} \mathsf{let}\ x = e;\ s} \ \mathrm{SS3.6}$$

$$\frac{\begin{array}{l}\Delta; \Upsilon; \Gamma \vdash_{\mathrm{rtyp}} e : \exists^{\phi} \alpha{:}\kappa.\tau' \\ \Delta, \alpha{:}\kappa; \Upsilon; \Gamma, x{:}\tau'; \tau \vdash_{\mathrm{styp}} s \\ \alpha \notin \mathrm{Dom}(\Delta) \quad x \notin \mathrm{Dom}(\Gamma) \\ \Delta \vdash_{\mathrm{k}} \tau : \mathtt{A}\end{array}}{\Delta; \Upsilon; \Gamma; \tau \vdash_{\mathrm{styp}} \mathsf{open}\ e\ \mathsf{as}\ \alpha, x;\ s} \ \mathrm{SS3.7} \qquad \frac{\begin{array}{l}\Delta; \Upsilon; \Gamma \vdash_{\mathrm{ltyp}} e : \exists^{\&} \alpha{:}\kappa.\tau' \\ \Delta, \alpha{:}\kappa; \Upsilon; \Gamma, x{:}\tau'*; \tau \vdash_{\mathrm{styp}} s \\ \alpha \notin \mathrm{Dom}(\Delta) \quad x \notin \mathrm{Dom}(\Gamma) \\ \Delta \vdash_{\mathrm{k}} \tau : \mathtt{A}\end{array}}{\Delta; \Upsilon; \Gamma; \tau \vdash_{\mathrm{styp}} \mathsf{open}\ e\ \mathsf{as}\ \alpha, *x;\ s} \ \mathrm{SS3.8}$$

Figure 3.7: Chapter 3 Typing, Statements

$$\frac{\Upsilon; x\cdot \vdash \text{gettype}(\Gamma(x), p, \tau) \quad \Delta \vdash_{\text{k}} \Gamma(x) : \mathtt{A} \quad \vdash_{\text{wf}} \Delta; \Upsilon; \Gamma}{\Delta; \Upsilon; \Gamma \vdash_{\text{ltyp}} xp : \tau} \text{ SL3.1}$$

$$\frac{C \vdash_{\text{rtyp}} e : \tau* \quad \Delta \vdash_{\text{k}} \tau : \mathtt{A}}{C \vdash_{\text{ltyp}} *e : \tau} \text{ SL3.2} \qquad \frac{C \vdash_{\text{ltyp}} e : \tau_0 \times \tau_1}{C \vdash_{\text{ltyp}} e.0 : \tau_0} \text{ SL3.3} \qquad \frac{C \vdash_{\text{ltyp}} e : \tau_0 \times \tau_1}{C \vdash_{\text{ltyp}} e.1 : \tau_1} \text{ SL3.4}$$

$$\frac{\Upsilon; x\cdot \vdash \text{gettype}(\Gamma(x), p, \tau) \quad \Delta \vdash_{\text{k}} \Gamma(x) : \mathtt{A} \quad \vdash_{\text{wf}} \Delta; \Upsilon; \Gamma}{\Delta; \Upsilon; \Gamma \vdash_{\text{rtyp}} xp : \tau} \text{ SR3.1}$$

$$\frac{C \vdash_{\text{rtyp}} e : \tau* \quad \Delta \vdash_{\text{k}} \tau : \mathtt{A}}{C \vdash_{\text{rtyp}} *e : \tau} \text{ SR3.2} \qquad \frac{C \vdash_{\text{rtyp}} e : \tau_0 \times \tau_1}{C \vdash_{\text{rtyp}} e.0 : \tau_0} \text{ SR3.3} \qquad \frac{C \vdash_{\text{rtyp}} e : \tau_0 \times \tau_1}{C \vdash_{\text{rtyp}} e.1 : \tau_1} \text{ SR3.4}$$

$$\frac{\vdash_{\text{wf}} C}{C \vdash_{\text{rtyp}} i : \text{int}} \text{ SR3.5} \qquad \frac{C \vdash_{\text{ltyp}} e : \tau}{C \vdash_{\text{rtyp}} \&e : \tau*} \text{ SR3.6} \qquad \frac{C \vdash_{\text{rtyp}} e_0 : \tau_0 \quad C \vdash_{\text{rtyp}} e_1 : \tau_1}{C \vdash_{\text{rtyp}} (e_0, e_1) : \tau_0 \times \tau_1} \text{ SR3.7}$$

$$\frac{\Delta; \Upsilon; \Gamma \vdash_{\text{ltyp}} e_1 : \tau \quad \Delta; \Upsilon; \Gamma \vdash_{\text{rtyp}} e_2 : \tau \quad \Delta \vdash_{\text{asgn}} \tau}{\Delta; \Upsilon; \Gamma \vdash_{\text{rtyp}} e_1 = e_2 : \tau} \text{ SR3.8}$$

$$\frac{C \vdash_{\text{rtyp}} e_1 : \tau' \to \tau \quad C \vdash_{\text{rtyp}} e_2 : \tau'}{C \vdash_{\text{rtyp}} e_1(e_2) : \tau} \text{ SR3.9} \qquad \frac{C; \tau \vdash_{\text{styp}} s \quad \vdash_{\text{ret}} s}{C \vdash_{\text{rtyp}} \mathsf{call}\ s : \tau} \text{ SR3.10}$$

$$\frac{\Delta; \Upsilon; \Gamma \vdash_{\text{rtyp}} e : \forall \alpha{:}\kappa.\tau' \quad \Delta \vdash_{\text{ak}} \tau : \kappa}{\Delta; \Upsilon; \Gamma \vdash_{\text{rtyp}} e[\tau] : \tau'[\tau/\alpha]} \text{ SR3.11}$$

$$\frac{\Delta; \Upsilon; \Gamma \vdash_{\text{rtyp}} e : \tau[\tau'/\alpha] \quad \Delta \vdash_{\text{ak}} \tau' : \kappa \quad \Delta \vdash_{\text{k}} \exists^{\phi}\alpha{:}\kappa.\tau : \mathtt{A}}{\Delta; \Upsilon; \Gamma \vdash_{\text{rtyp}} \mathsf{pack}\ \tau', e\ \mathsf{as}\ \exists^{\phi}\alpha{:}\kappa.\tau : \exists^{\phi}\alpha{:}\kappa.\tau} \text{ SR3.12}$$

$$\frac{\Delta; \Upsilon; \Gamma, x{:}\tau; \tau' \vdash_{\text{styp}} s \quad \vdash_{\text{ret}} s \quad x \notin \text{Dom}(\Gamma)}{\Delta; \Upsilon; \Gamma \vdash_{\text{rtyp}} (\tau\ x) \to \tau'\ s : \tau \to \tau'} \text{ SR3.13}$$

$$\frac{\Delta, \alpha{:}\kappa; \Upsilon; \Gamma \vdash_{\text{rtyp}} f : \tau \quad \vdash_{\text{wf}} \Delta; \Upsilon; \Gamma \quad \alpha \notin \text{Dom}(\Delta)}{\Delta; \Upsilon; \Gamma \vdash_{\text{rtyp}} \Lambda\alpha{:}\kappa.f : \forall \alpha{:}\kappa.\tau} \text{ SR3.14}$$

Figure 3.8: Chapter 3 Typing, Expressions

$$\frac{}{\Upsilon; xp \vdash \text{gettype}(\tau, \cdot, \tau)} \qquad \frac{\Upsilon; xp\mathtt{u} \vdash \text{gettype}(\tau'[\Upsilon(xp)/\alpha], p', \tau)}{\Upsilon; xp \vdash \text{gettype}(\exists^{\&}\alpha{:}\kappa.\tau', \mathtt{u}p', \tau)}$$

$$\frac{\Upsilon; xp0 \vdash \text{gettype}(\tau_0, p', \tau)}{\Upsilon; xp \vdash \text{gettype}(\tau_0 \times \tau_1, 0p', \tau)} \qquad \frac{\Upsilon; xp1 \vdash \text{gettype}(\tau_1, p', \tau)}{\Upsilon; xp \vdash \text{gettype}(\tau_0 \times \tau_1, 1p', \tau)}$$

Figure 3.9: Chapter 3 Typing, Heap Objects

$$\frac{}{\vdash_{\text{ret}} \mathsf{return}\ e} \qquad \frac{\vdash_{\text{ret}} s_1 \quad \vdash_{\text{ret}} s_2}{\vdash_{\text{ret}} \mathsf{if}\ e\ s_1\ s_2} \qquad \frac{\vdash_{\text{ret}} s}{\begin{array}{l}\vdash_{\text{ret}} s; s' \quad \vdash_{\text{ret}} \mathsf{let}\ x = e;\ s \\ \vdash_{\text{ret}} s'; s \quad \vdash_{\text{ret}} \mathsf{open}\ e\ \mathsf{as}\ \alpha, x;\ s \\ \qquad\qquad \vdash_{\text{ret}} \mathsf{open}\ e\ \mathsf{as}\ \alpha, *x;\ s\end{array}}$$

Figure 3.10: Chapter 3 Must-Return

$$\frac{}{\Upsilon; \Gamma \vdash_{\text{htyp}} \cdot : \cdot} \qquad \frac{\Upsilon; \Gamma \vdash_{\text{htyp}} H : \Gamma' \quad \cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v : \tau}{\Upsilon; \Gamma \vdash_{\text{htyp}} H, x \mapsto v : \Gamma', x{:}\tau}$$

$$\frac{}{H \vdash_{\text{refp}} \cdot} \qquad \frac{H \vdash_{\text{refp}} \Upsilon \quad \text{get}(H(x), p, \mathsf{pack}\ \tau', v\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau)}{H \vdash_{\text{refp}} \Upsilon, xp{:}\tau'}$$

$$\frac{\Upsilon; \Gamma \vdash_{\text{htyp}} H : \Gamma \quad H \vdash_{\text{refp}} \Upsilon \quad \cdot; \Upsilon; \Gamma; \tau \vdash_{\text{styp}} s \quad \vdash_{\text{ret}} s}{\vdash_{\text{prog}} H; s}$$

Figure 3.11: Chapter 3 Typing, States

Type-checking also restricts what types can appear where, using the judgments in Figure 3.6. The $\vdash_{\text{ak}}$ and $\vdash_{\text{wf}}$ judgments primarily ensure that type variables are in scope. The $\vdash_{\text{k}}$ kinding judgment forbids abstract types except under pointers. We use it to prevent manipulating terms of unknown size, although formalizing this restriction is somewhat contrived because the dynamic semantics for the formal machine would have no trouble allowing terms of unknown size. The $\vdash_{\text{asgn}}$ judgment describes types of mutable expressions.

We do not need the judgments in Figure 3.11 to check source programs. They describe the invariant we need to prove type safety in the next section. If s is allowed as a source program, then $\vdash_{\text{prog}} \cdot; s$.

We now describe the judgments in more detail.

If $\Delta \vdash_{\text{k}} \tau : \kappa$, then given the type variables in Δ, type τ has kind κ and its size is known. To prevent types of unknown size, we cannot derive $\Delta, \alpha{:}\mathtt{A} \vdash_{\text{k}} \alpha : \kappa$, but we can derive $\Delta, \alpha{:}\mathtt{A} \vdash_{\text{k}} \alpha* : \mathtt{B}$. For simplicity, we assume function types have known size, unlike in Cyclone. We can imagine implementing all function definitions with values of the same size (e.g., pointers to code), so this simplification is justifiable. Some types are not subject to the known-size restriction, such as τ in $e[\tau]$. But we still require $\Delta \vdash_{\text{ak}} \tau : \kappa$; we can derive $\Delta, \alpha{:}\mathtt{A} \vdash_{\text{ak}} \alpha : \kappa$. The types for which $\Delta \vdash_{\text{asgn}} \tau$ have known size *and* any types of the form $\exists^{\&}\alpha{:}\kappa'.\tau$ occur under pointers.

We cannot give quantified types kind B, but we argued earlier that doing so

is not useful. We exploit this fact in the rules for $\vdash_{\text{asgn}}$: It is too lenient to allow $\Delta, \alpha{:}\mathtt{B} \vdash_{\text{asgn}} \alpha$ if we might instantiate α with a type of the form $\exists^{\&}\alpha{:}\kappa'.\tau$. We could enrich the kind system to distinguish assignable box kinds and unassignable box kinds (the former being a subkind of the latter), but again it is not useful.

Well-formed contexts (the $\vdash_{\text{wf}}$ judgments) have only known-size types without free type variables. Because Υ is used only to describe heaps, no Δ is necessary.

The typing rules for statements are unsurprising, so we describe only some of them. Rule SS3.2 uses the τ in the context to ensure functions do not return values of the wrong type. In rule SS3.6, the body of the binding is checked in an extended context, as usual. Rules SS3.7 and SS3.8 allow the two forms of existential unpacks. As expected, they extend Δ and Γ and the type of the bound term variable depends on the form of the unpack (τ' in SS3.7 and $\tau'*$ in SS3.8). The reuse of α in the type of e is not a restriction because existential types α-convert. The e in SS3.8 must be a valid left-expression, so we type-check it with $\vdash_{\text{ltyp}}$, as opposed to $\vdash_{\text{rtyp}}$ in SS3.7. The type of e in SS3.8 cannot have the form $\exists^{\delta}\alpha{:}\kappa.\tau$; this is the essence of the restriction on such types. Finally, the kinding assumption in SS3.7 and SS3.8 is a technical point to ensure that τ does not have a free occurrence of α, which is always possible by α-conversion of the open statement.

Note that my previous work [94, 93] has a minor error: It does not enforce that e in open e as $\alpha, *x$; s is a valid left expression. In terms of that work, the accidentally omitted assumption (assumed in the type-safety proof) is $\vdash e$ lval.

The rules for $\vdash_{\text{ltyp}}$ are a subset of the rules for $\vdash_{\text{rtyp}}$. We could have restricted the form of left-expressions more directly and used just one conventional type-checking judgment for all expressions. In subsequent chapters, the rules for valid left expressions are more lenient than a syntactic restriction of valid right expressions, so for uniformity this chapter uses a separate judgment. A syntactic restriction suffices in this chapter because programs always have "read access" of all data. In subsequent chapters, we reject e as a right-expression if the program does not have access to e, but we allow it as a left-expression because $\&e$ does not access e.

We now describe the type-checking rules for right-expressions. To type-check xp, SR3.1 uses the gettype relation to derive a type from the type of x and the form of p. We can use u to acquire the contents of an existential package only if the package has a type of the form $\exists^{\&}\alpha{:}\kappa.\tau$. Such types are not assignable, so no mutation can interfere. Furthermore, to use u, the path to the package must be in Υ. We use Υ to remember the witness types of all packages that have been unpacked with a statement of the form open e as $\alpha, *x$; s. These witnesses cannot change, so it is sound to use $\Upsilon(xp)$. Before a program executes, no packages have been unpacked, so Υ is $\cdot$. In fact, there is no need for gettype at all in source programs because we can forbid nonempty paths. SR3.2 prevents dereferencing a pointer to a value of unknown size. SR3.3–7 hold no surprises. SR3.8 ensures that

e_1 is a valid left-expression and its type is assignable. SR3.9 is the normal rule for function call. SR3.10 requires $\vdash_{\text{ret}} s$, so we can prove that execution cannot produce stuck terms of the form $\mathsf{call}\ v$. SR3.11 and SR3.12 are conventional for quantified types. We use the $\vdash_{\text{ak}}$ judgment because types for instantiations and witnesses can have unknown size. SR3.13 and SR3.14 ensure functions return and assume the correct kinds for quantified types. Unlike C and Cyclone, we do not require that functions are closed (modulo global variables) nor do we require that they appear at top-level.

The rules for $\vdash_{\text{ret}} s$ are all straightforward. All terminating statements become either v or $\mathsf{return}\ v$ for some v. The $\vdash_{\text{ret}} s$ judgment is a conservative analysis that forbids the former possibility.

The judgment $\vdash_{\text{prog}} H; s$ describes the invariant we use to establish type safety. First, the heap must type-check without reference to any free variables or any type variables. By checking $\Gamma; \Upsilon \vdash_{\text{htyp}} H : \Gamma$, we allow mutually recursive functions in the heap. (Mutually recursive data has to be encoded with functions because we do not have recursive types.) Second, if $\Upsilon(xp) = \tau$, then the value in the heap location that xp describes has to be an existential package with witness type τ, and the package's type must indicate that the witness will not change. Third, s has to type-check under the Γ and Υ that describe the heap. Finally, we require $\vdash_{\text{ret}} s$, though it does not really matter.

3.5.4 Type Safety

Appendix A proves this result:

Definition 3.1. *State $H; s$ is <u>stuck</u> if s is not of the form $\mathsf{return}\ v$ and there are no H' and s' such that $H; s \xrightarrow{\text{s}} H'; s'$.*

Theorem 3.2 (Type Safety). *If $\cdot; \cdot; \cdot; \tau \vdash_{\text{styp}} s$, $\vdash_{\text{ret}} s$, and $\cdot; s \xrightarrow{\text{s}}{}^{*} H'; s'$ (where $\xrightarrow{\text{s}}{}^{*}$ is the reflexive, transitive closure of $\xrightarrow{\text{s}}$), then $H'; s'$ is not stuck.*

Informally, well-typed programs can continue evaluating until they terminate (though they may not terminate).

3.6 Related Work

The seminal theoretical foundation for quantified types in programming languages is the polymorphic lambda calculus, also called System F, which Girard [87] and Reynolds [177] invented independently. Many general-purpose programming languages, most notably Standard ML [149], OCaml [40, 141], and Haskell [130] use quantified types and type constructors to allow code reuse.

Higher level languages generally do not restrict the types that a type variable can represent. A polymorphic function can be instantiated at any type, including records and floating-point types. Simpler implementations add a level of indirection for all records and floating-point numbers to avoid code duplication. Sophisticated analyses and compiler intermediate languages can avoid some unnecessary levels of indirection [154, 195, 139, 140, 217]. In the extreme, ML's lack of polymorphic recursion lets whole-program compilers *monomorphize* the code, essentially duplicating polymorphic functions for each type at which they are instantiated [152, 21]. The amount of generated code appears tolerable in practice. C++ [193] defines template instantiation in terms of code duplication, making template functions closer to advanced macros than parametric polymorphism.

An example of a simple compromise is the current OCaml implementation [220]: Records and arrays of floating-point numbers do not add a level of indirection for the numbers. Polymorphic code for accessing an array (in Cyclone terms, something of type α`[]`), must check at run-time whether the array holds floating-point numbers or not, so run-time type information is necessary.

Without first-class polymorphism or polymorphic recursion, ML and Haskell enjoy full type inference: Programs never need explicit type information. Type inference is undecidable (it is uncomputable whether a term without explicit types can type-check) if we add first-class polymorphism or polymorphic recursion [216, 114, 133]. Haskell 98 [130] includes polymorphic recursion, but requires explicit types for functions that use it. Because these languages encourage using many functions, conventional wisdom considers Cyclone's approach of requiring explicit types for all function definitions intolerable. However, room for compromise between inference and more powerful type systems exists, as proposals for ML extensions and additions to Haskell implementations demonstrate [83, 174, 197, 196].

Section 3.4.2 described how bounded quantification for types could increase the Cyclone type system's expressiveness. The type theory for bounded quantification has received considerable attention, particularly because of its role in encoding some object-oriented idioms [33]. An important negative result concerns bounded quantification's interaction with subtyping: It is sound to consider $\forall\alpha \leq \tau_1.\tau_2$ a subtype of $\forall\alpha \leq \tau_3.\tau_4$ if τ_3 is a subtype of τ_1 and τ_2 is a subtype of τ_4. However, together with other conventional subtyping rules, this rule for subtyping universal types makes the subtyping question (i.e., given two types, is one a subtype of the other) undecidable [172]. A common compromise is to require equal bounds ($\tau_1 = \tau_3$ in our example) [37]. Another possibility is to require explicit subtyping proofs (or hints about proofs) in source programs.

The problem with polymorphic references discussed in Section 3.3 has received much attention from the ML community [198, 219, 108]. In ML, a commitment to

full type inference and an advanced module system with abstract types complicate the problem. So-called "weak type variable" solutions, which make a kind distinction with respect to mutation, have fallen out of favor. Instead, a simple "value restriction" suffices. Essentially, a binding cannot receive a universal type unless it is initialized with a syntactic value, such as a variable (which is immutable) or a function definition. This solution interacts well with type inference and appears tolerable in practice. In Cyclone, more explicit typing makes the solution of forbidding type instantiation in left-expressions seem natural.

Explicit existential types have not been used as much in designing programming languages. Mitchell and Plotkin's seminal work [151] showed how constructs for abstract types, such as the `rep` types in CLU clusters [144] and the `abstype` declarations in Standard ML [149] are really existential types. Encodings of closures [150] and objects [33] using existential types suggest that the lack of explicit existential types in many languages is in some sense an issue of terminology. Current Haskell implementations [197, 196] include existential types for "first-class" values, as suggested by Läufer [137]. In all the above work, existential packages are immutable, so the problem from Section 3.3 is irrelevant.

Other lower-level typed languages have included existential types, but have not encountered the same unsoundness problem. For example, Typed Assembly Language [157] does not have a way to create an alias of an opened type, as with Cyclone's reference patterns. There is also no way to change the type of a value in the heap—assigning to an existential package means making a pointer refer to a different heap record. Xanadu [222], a C-like language with compile-time reasoning about integer values, also does not have aliases at the opened type. Roughly, `int` is short-hand for $\exists\alpha{:}\mathtt{I}.\alpha$ and uses of `int` values implicitly include the necessary `open` expressions. This expression *copies* the value, so aliasing is not a problem. It appears that witness types can change because mutating a heap-allocated `int` would change its witness.

Languages with *linear* existential types can provide a solution different than the ones presented in this work. In these systems, there is only one reference to an existential package, so *a fortiori* there are no aliases at the opened type. Walker and Morrisett [212] exploit this invariant to define `open` such that it does not introduce any new bindings. Instead, it mutates the location holding the package to hold the package's contents. Without run-time type information, such an `open` has no actual effect. The Vault system [55] also has linear existential types. Formally, opening a Vault existential package introduces a new binding. In practice, the Vault type-checker infers where to put `open` and `pack` terms and how to rewrite terms using the bindings that `open` statements introduce. This inference may make Vault's existential types more convenient.

Section 3.4 suggested extending Cyclone with a way for programs to use run-time tests to refine information about an unknown type safely. An apparent disadvantage of such an extension is that it would violate *parametricity*, a well-known concept for reasoning about the behavior of polymorphic functions [192, 178, 146, 205]. As a simple example, in the polymorphic lambda calculus, a term with the type $\forall\alpha\forall\beta.(\alpha \times \beta) \rightarrow (\beta \times \alpha)$ must behave equivalently to the function that given (e_0, e_1) returns (e_1, e_0). However, Pierce and Sangiorgi [173] presented a very clever trick showing that languages with mutable references (such as ML) can violate parametricity. Morrisett, Zdancewic, and I [99] argued that the true source of the ability to violate parametricity is aliasing of values at more and less abstract types (e.g., a value available at types α`*` and `int*`). Recent work by Naumann and Banerjee [18] has restricted aliasing to establish parametricity in a setting with mutation. Because Cyclone does not restrict aliasing, the type system does not ensure parametricity. Instead, it ensures only basic memory safety.

The Typed Assembly Language implementation [155] for the IA-32 architecture has a more powerful kind system than Cyclone, though the details are not widely known. For each number i, there is a kind `M`i describes types of memory objects consuming i bytes. These kinds are subkinds of `M`, which corresponds to kind `A` in Cyclone. At the assembly level, padding and alignment are explicit, so giving types these more descriptive kinds is more appropriate. However, the fine granularity of assembly-language instructions make it difficult for the type system to allow safe use of an abstract value. For example, given a pointer to a value of type α of kind `M12`, we might like to push a copy of the pointed to value onto the stack. Doing so requires adjusting the stack pointer by 12 bytes and executing multiple move instructions for the parts of the abstract value. I do not believe the details for allowing such an operations were ever implemented.

The GHC [196] Haskell implementation provides alternate forms of floating-point numbers and records that do not have extra levels of indirection. Their uses are even more restricted than in Cyclone. Not only do values of these types essentially have kind `A` in a language without type variables of kind `A`, but unboxed records can appear only in certain syntactic positions. Nonetheless, these extensions let programmers control data representation enough to improve performance for certain applications.

There has been remarkably little work on quantified types for C-like languages. Smith and Volpano [187, 188] describe an integration of universal types with C. Their formal development has some similarities with my work, but they do not consider `struct` types. Therefore, they have no need for existential types.

Type quantification is not the only way to prohibit unsafe casts from `void*`. Chapter 8 discusses other approaches.

Chapter 4

Region-Based Memory Management

Cyclone uses *region-based* memory-management to prevent dangling-pointer dereferences. Every memory object is in exactly one region and all of a region's objects are deallocated simultaneously. To avoid run-time overhead, the system encodes lifetime information in the type system. Despite imposing more memory-management structure than C, the system allows many important idioms. It integrates C-style stack allocation, last-in-first-out regions of unbounded size, and an immortal heap that allows implicit conservative garbage collection. Usually the same code can operate on objects regardless of where they are allocated.

This range of options is an important step toward Cyclone's goals. We provide more control over memory management than safe high-level languages, without sacrificing safety, resorting to hidden run-time sate, or requiring code duplication. More specifically, the system for preventing dangling-pointer dereferences is:

- *Sound:* Programs never dereference dangling pointers.
- *Static:* Dereferencing a dangling pointer is a compile-time error. We do not use run-time checks to determine if memory has been deallocated.
- *Convenient:* We minimize the need for explicit programmer annotations while supporting many C idioms. In particular, many uses of the addresses of local variables require no modification.
- *Exposed:* Programmers control where objects are allocated and how long they live. As usual, all local variables are stack-allocated.
- *Comprehensive:* We treat all memory uniformly, including the stack, the heap (which can optionally be garbage-collected), and "growable" regions.

- *Scalable:* The system supports separate compilation because all analyses are intraprocedural.

Section 4.1 describes the basic techniques used to achieve these design goals. Section 4.2 describes the interaction between the region system and quantified types. The critical issue is interacting with data-hiding constructs (existential packages) that might have dangling pointers not reflected in their type. The Cyclone solution makes existential types a bit less convenient, but the type information for code not using existential types remains simple. Section 4.3 describes the simple run-time support necessary for the region system.

Compared to C, the language imposes more restrictions (e.g., one cannot call the `free` function) and requires more explicit type information. Section 4.4 describes informally the strengths of the region system and what extensions would be needed to capture additional idioms. Many of these extensions are already experimental parts of Cyclone, but this dissertation does not cover them in depth. The region system presented here is a relatively mature aspect of Cyclone that has been used extensively. Previously published work [97] (from which this chapter borrows heavily) measures the programmer burden and performance cost relative to C code. These measurements corroborate my subjective evaluation.

Section 4.5 and Appendix B present a formal abstract machine with region-based memory management and prove that its type system is safe. For this machine, safety implies that objects are not accessed after they are deallocated. Compared with the abstract machine in Chapter 3, the heap has more structure precisely because objects are in regions.

As discussed in Section 4.6, Cyclone is not the first system to include region information in its type system. However, as an explicitly typed, low-level language designed for human programmers, it does make several technical contributions explained in this chapter:

- *Region subtyping:* A last-in-first-out discipline on region lifetimes induces an "outlives" relationship on regions, which lets us provide a useful subtyping discipline on pointer types.

- *Simple effects:* We eliminate the need for effect variables (which complicate interfaces) by using the novel "`regions(`τ`)`" type operator.

- *Default annotations:* We combine a local inference algorithm with a system of defaults to reduce the need for explicit region annotations.

- *Integration of existential types:* The combination of region subtyping and simple effects makes the integration of first-class abstract types relatively simple.

Readers familiar with previous work on Cyclone's regions [97] may wish to focus on Sections 4.4, 4.5, and Appendix B because the other sections are just revisions. However, Section 4.6 gives a more detailed description of related work.

4.1 Basic Constructs

This section presents the basic features of Cyclone's memory-management system. It starts with the constructs for creating regions, allocating objects, and so on—this part is simple because the departure from C is small. We next present the corresponding type system, which is more involved because every pointer type carries a region annotation. We exploit quantified types and type constructors to avoid committing to particular regions, just as terms in Chapter 3 avoid committing to particular types. Then we show how regions' lifetimes induce subtyping on pointer types. At that point, the type syntax is quite verbose, so we explain the features that, in practice, eliminate most region annotations.

4.1.1 Region Terms

In Cyclone, all memory is in some region, of which there are three flavors:

- A single heap region, which conceptually lives forever
- Stack regions, which correspond to local-declaration blocks, as in C
- Dynamic regions, which have lexically scoped lifetimes but permit unlimited allocation into them

Static data objects reside in the heap. Primitives `malloc` and `new` create new heap objects. The `new` operation is like `malloc` except that it takes an expression and initializes the memory with it. There is no explicit mechanism for reclaiming heap-allocated objects (e.g., `free`). However, Cyclone programs can link against the Boehm-Demers-Weiser conservative garbage collector [26] to reclaim unreachable heap-allocated objects. Section 4.3 discusses the interaction between the collector and regions.

Stack regions correspond to C's local-declaration blocks: entering a block with local declarations creates storage with a lifetime corresponding to the lexical scope of the block. Function parameters are in a stack region corresponding to the function's lifetime. In short, Cyclone local declarations and function parameters have the same layout and lifetime as in C.

Dynamic regions are created with the construct `region r;` s, where `r` is an identifier and s is a statement. The region's lifetime is the execution of s. In s,

`r` is bound to a region *handle*, which primitives `rmalloc` and `rnew` use to allocate objects into the associated region. For example, `rnew(r) 3` returns a pointer to an `int` allocated in the region of handle `r` and initialized to 3. Handles are first-class values; a caller may pass a handle to a function so it can allocate into the associated region. A predefined constant `heap_region` is a handle for the heap, so `new` and `malloc` are just short-hand for using `heap_region` with `rnew` and `rmalloc`.

Like a declaration block, a dynamic region is deallocated when execution leaves the body of the enclosed statement. Execution can leave due to unstructured jumps (`continue`, `goto`, etc.), a `return`, or via an exception. Section 4.3 explains how we compile dynamic-region deallocation.

The region system imposes no changes on the representation of pointers or the meaning of operators such as `&` and `*`. There are no hidden fields or reference counts for maintaining region information at run-time. The infrastructure for preventing dangling-pointer dereferences is in the type system, making such dereferences a compile-time error.

4.1.2 Region Names

Ignoring subtyping, all pointers always point into exactly one region. Pointer types include the *region name* of the region they point into. For example, `int*`ρ describes a pointer to an `int` that is in the region named is ρ. The invariant that pointers have a particular region is the basic restriction we impose to make the undecidable problem of detecting dangling-pointer dereferences tractable. Pointer types with different region names are different types. A handle for a region corresponding to ρ has the type `region_t<`ρ`>`. Were it not for subtyping, handle types would be *singletons*: two handles with the same type would be the same handle.

Region names fall into three flavors, corresponding to the three region flavors. The region name for the heap is ρ_H. A block labeled `L` (e.g., `L:{int x=0;`s`}`) has name ρ_L and refers to the stack region that the block creates. Considering a function definition a labeled block, a function named `f` has a region named ρ_f in which the parameters are allocated. Finally, the statement `region r;` s defines region name ρ_r for the created region. So `r` has type `region_t<`ρ_r`>`. In all cases, the scope of a region name corresponds to the lifetime of the corresponding region.

We can now give types to some examples. If e_1 has type `region_t<`ρ`>` and e_2 has type τ, then `rnew (`e_1`)` e_2 has type τ`*`ρ. If `int x` is declared in block `L`, then `&x` has type `int*`ρ_L. Similarly, if e has type τ`*`ρ, then `&*`e has type τ`*`ρ.

To dereference a pointer, safety demands that its region be live. Our goal is to determine at compile-time that no code follows a dangling pointer. It often suffices to ensure that pointer types' region names are in scope. For example, this code is ill-typed:

```
int*ρ_L p;
L:{ int x = 0;
     p = &x; }
*p = 42;
```

The code creates storage for `x` that is deallocated before the last line, so the assignment of `&x` to `p` creates a dangling pointer that the last assignment dereferences. Cyclone rejects this code because ρ_L is not in scope when `p` is declared. If we change the declaration of `p` to use another region name, then the assignment `p = &x` fails to type-check because `&x` has type `int*`ρ_L.

However, Cyclone's existential types allow pointers to escape the scope of their regions, just as closures do in functional languages [201]. Therefore, in general, we cannot rely on simple scoping mechanisms to ensure soundness. Instead, we must track the set of live region names at each control-flow point. To keep the analysis intraprocedural, we use a novel type-and-effects system to track interprocedural liveness requirements. We delay the full discussion of effects until Section 4.2.

To understand the correct region name for a pointer type, it helps to emphasize that left-expressions have types and region names. In the example above, `&x` has type `int*`ρ_L because the left-expression `x` has type `x` and region name ρ_L. Similarly, if e is a right-expression with type $\tau * \rho$, then $*e$ is a left-expression with region name ρ and an assignment of the form $*e = e'$ is safe only if the region named ρ is live. Section 4.5 describes the type-checking rules for left-expressions precisely.

4.1.3 Quantified Types and Type Constructors

Region names are type variables that describe regions instead of terms. The kind system distinguishes region names from other type variables: A region name has kind `R`, which is incomparable to the kinds `B` and `A` that describe ordinary types. Because region names are type variables, we can define region-polymorphic functions, abstract types that hide region names, and type constructors with region-name parameters. This section demonstrates that these natural features are extremely important for the expressiveness of the Cyclone region system. In particular, region polymorphism is much more common than type polymorphism.

Universal Quantification Functions in Cyclone are region-polymorphic; they can abstract the actual regions of their arguments or results. That way, functions can manipulate pointers regardless of whether they point into the stack, the heap, or a dynamic region. For example, in this contrived program, `fact` abstracts a region name ρ and takes a pointer into the region named ρ:

```
void fact<ρ>(int*ρ result, int n) {
 L: { int x = 1;
      if(n > 1) fact<ρ_L>(&x,n-1);
      *result = x*n; }
}
int g = 0;
int main() { fact<ρ_H>(&g,6); return g; }
```

When executed, the program returns the value 720. In `main`, we pass `fact` a heap pointer (`&g`), so the type of `fact` is instantiated with ρ_H for ρ. Each recursive call instantiates ρ with ρ_L, the name of the local stack region. This polymorphic recursion allows us to pass a pointer to the locally declared variable `x`. At run time, the first instance of `fact` modifies `g`; each recursive call modifies its caller's stack frame. Alternatively, we could have written the function as:

```
  void fact2<ρ>(int*ρ result, int n) {
  if(n > 1) fact2<ρ>(result,n-1);
  *result *= n;
}
```

Here is a third version that uses a dynamic region to hold all of the intermediate results:

```
void fact3<ρ>(region_t<ρ> r,int*ρ result,int n) {
  int*ρ x = rnew(r) 1;
  if(n > 1) fact3<ρ>(r,x,n-1);
  *result = (*x)*n;
}
int main() {
  region r;
  int*ρ_r g = rnew(r) 0;
  return fact3<ρ_r>(r, g, 6);
}
```

The function `main` creates a dynamic region with handle `r` and uses `rnew(r) 0` to allocate an initial result pointer. Next, it calls `fact3`, instantiating ρ with ρ_r and passing the handle. Instead of stack-allocation, `fact3` uses the dynamic region to hold each recursive result, consuming space proportional to `n`. The space is reclaimed when control returns to `main`.

By using the same region name, function prototypes can assume and guarantee region equalities of unknown regions. In the examples below, `f1` does not type-check because it might assign a pointer into the wrong region:

```
void f1<ρ1,ρ2,ρ3>(int*ρ1*ρ2 pp, int*ρ3 p) { *pp = p; } // rejected
void f2<ρ1,ρ2>(int*ρ1*ρ2 pp, int*ρ1 p) { *pp = p; } // accepted
```

Region equalities are crucial for return types, particularly when the return value is placed in a caller-specified region:

```
int*ρ identity<ρ>(int*ρ p) { return p; }
int*ρ newzero<ρ>(region_t<ρ> h) { return rnew(h) 0; }
```

For example, `newzero<`ρ_H`>(heap_region)` has type `int*`ρ_H, which ensures the caller that the pointed-to object will conceptually live forever.

More realistic code also uses region polymorphism. For example, ignoring array-bounds, nul-terminators (strings ending with `'\0'`), and `NULL` pointers, the Cyclone string library provides prototypes like these:

```
char*ρ  strcpy<ρ,ρ2>(char*ρ d, const char*ρ2 s);
char*ρH strdup<ρ>(const char*ρ s);
char*ρ  rstrdup<ρ,ρ2>(region_t<ρ>,const char*ρ2 s);
int strlen<ρ>(const char*ρ s);
```

Parametricity ensures `strcpy` returns a pointer somewhere into its first argument.

Of course, not all functions are region polymorphic, as this example shows:

```
int*ρH g = NULL;
void set_g(int*ρH x) { g = x; }
```

Existential Quantification We can use existential quantification over region names to relate the regions for pointers and handles, as this example demonstrates:

```
struct T1 { <ρ1>
  int *ρ1*ρH p1;
  int *ρ1*ρH p2;
  region_t<ρ1> r;
};
```

Given a value of type `struct T1`, we might like to swap the contents of what `p1` and `p2` point to or mutate them to fresh locations allocated with `r`. However, `struct T1` is actually *useless* in the sense that no Cyclone program can use `r`, `**p1`, or `**p2`. As explained in Section 4.2, the region named ρ_1 may have been deallocated in which case such accesses are unsound. We will strengthen the definition of `struct T1` and the existential-type definitions from Chapter 3 to make them useful.

Type Constructors Because `struct` definitions can contain pointers, Cyclone allows these definitions to take region-name parameters. For example, here is a declaration for lists of pointers to ints:

```
struct RLst<ρ1,ρ2> {
  int*ρ1 hd;
  struct RLst<ρ1,ρ2> *ρ2 tl;
};
```

Ignoring subtyping, a value of type `struct RLst<`ρ_1`,`ρ_2`>` is a list with `hd` fields that point into ρ_1 and `tl` fields that point into ρ_2. Other invariants are possible: If the type of `tl` were `struct RLst<`ρ_2`,`ρ_1`>*` ρ_2, the declaration would describe lists where the regions for `hd` and `tl` alternated at each element.

Type abbreviations using `typedef` can also have region parameters. For example, we can define region-allocated lists of heap-allocated pointers with:

```
typedef struct RLst<ρH,ρ> *ρ list_t<ρ>;
```

4.1.4 Subtyping

If the region corresponding to ρ_1 *outlives* the region corresponding to ρ_2, then it is sound to cast from type τ`*`ρ_1 to type τ`*`ρ_2. The last-in-first-out region discipline makes such outlives relationships common: when we create a region, we know every region currently live will outlive it. For example, a local variable can hold different function arguments:

```
void f<ρ1, ρ2>(int b, int*ρ1 p1, int*ρ2 p2) {
 L: { int*ρL p;
      if(b) p=p1; else p=p2;
      /* ... use p ... */ }
}
```

Without subtyping, the program fails to type-check because neither `p1` nor `p2` has type `int*`ρ_L. If we change the type of `p` to `int*`ρ_1 or `int*`ρ_2, then one of the assignments is illegal. With subtyping, both assignments use subtyping to cast (implicitly) to τ`*`ρ_L.

To ensure soundness, we do not allow casting τ_1`*`ρ to τ_2`*`ρ, even if τ_1 is a subtype of τ_2, as this cast would allow putting a τ_2 in a location where other code expects a τ_1. (This problem is the usual one with covariant subtyping on references.) However, we can allow casts from τ_1`*`ρ to `const` τ_2`*`ρ when τ_1 is a subtype of τ_2, if we enforce read-only access for `const` values (unlike C). This support for

"deep" subtyping, when combined with polymorphic recursion, is powerful enough to allow stack allocation of some structures of arbitrary size.

Intraprocedurally, the "created region outlives all live regions" rule suffices to establish outlives relationships. If the safety of a function requires that some arguments have an outlives relationship, then the function must have an explicit constraint that expresses a partial order on region lifetimes. The constraint, which is part of the function's type, is assumed when type-checking the function body and is a precondition for calling the function. Here is a simple example:

`void set<`ρ_1`,`ρ_2`>(int*`ρ_1`*`ρ_H `x, int*`ρ_2`*`ρ_H `y :` ρ_1`<`ρ_2`) { *y=*x; }`

The constraint $\rho_1<\rho_2$ indicates that the region named ρ_1 outlives the region named ρ_2.

4.1.5 Default Annotations

Cyclone employs a combination of carefully chosen defaults and intraprocedural inference to reduce dramatically the amount of necessary explicit type information. Because region names are type variables, many of the rules in Section 3.1.4 apply to them. Every pointer type includes a (possibly implicit) region name, so these rules gain importance compared to Chapter 3. The rules are slightly different for region names, as this section explains.

Due to type inference within function bodies, implicit type instantiation of polymorphic functions, and implicit subtyping, Cyclone programmers rarely write region names within function bodies. In particular, region names for local declaration blocks are almost never used explicitly. Given a block without an explicit label, the type-checker just creates a region name. The explicit labels and type instantiations in previous examples were for expository purposes only.

Because function definitions and function prototypes at the top-level implicitly universally quantify over free type variables (including region names), all of the explicit bindings in previous examples in this chapter are unnecessary. As before, explicit bindings are necessary only for first-class polymorphism. Even when explicitly bound, the type-checker infers the kind of a type variable based on its uses in the argument and result types.

Furthermore, the type-checker fills in omitted region names in function argument types with fresh type variables of region kind. For function result types, a fresh type variable is not a good default because a region-polymorphic function that could return a pointer into any region could return only `NULL`. Therefore, in function return types, the default region name is ρ_H.

Given these rules, programs like our first `fact` example need no region annotations:

```
void fact(int* result, int n) {
 int x = 1;
 if(n > 1) fact(&x,n-1);
 *result = x*n;
}
int g = 0;
int main() { fact(&g,6); return g; }
```

In other words, the code is a C program that ports to Cyclone without modification.

More generally, explicit annotations are necessary only to express region equalities on which safety relies. For example, if we write:

```
void f2(int** pp, int* p) {*pp=p;}
```

then the code elaborates to:

```
void f2<ρ1,ρ2,ρ3>(int *ρ1*ρ2 pp, int *ρ3 p) {*pp=p;}
```

which fails to type-check because `int*`$\rho_1 \neq$ `int*`ρ_3. The programmer must insert an explicit region annotation to assert an appropriate equality relation on the parameters:

```
void f2(int*ρ* pp, int*ρ p) { *pp = p; }
```

For more realistic examples, here are the string-library prototypes presented earlier but without unnecessary annotations:

```
char*ρ  strcpy(char*ρ d, const char* s);
char* strdup(const char* s);
char*ρ  rstrdup(region_t<ρ>,const char* s);
int strlen(const char* s);
```

The default rules for type definitions are not as convenient. The type-checker uses ρ_H in place of omitted region names. Type variables (including region names) must be explicitly bound. For example, the `struct Lst` example above cannot have any annotations removed. Fortunately, type definitions usually account for a small portion of a program's text.

Abstract and recursive `struct` definitions make it difficult to take a `struct` definition with omitted region annotations and implicitly make it a type constructor taking arguments for automatically filled in region names. First, for abstract types such rules make no sense because the field definitions are not available. Hence when providing an abstract interface, programmers would have to give explicit type-constructor parameter names and kinds anyway. Second, with recursive

(or mutually recursive) types it is not clear how many parameters a type constructor should have. Naively generating a fresh region name everywhere one is omitted would require an infinite number of region names for a definition like `struct Lst { int* hd; struct Lst *tl; };`. Another complication is that type constructors sometimes require explicit instantiation, so we would need rules on the order of the inferred parameters. However, default rules such as regularity (assuming recursive instances are instantiated with the same arguments) are a recent addition to Cyclone.

Although defining type constructors requires explicit region names, using them often does not. We can partially apply parameterized type definitions; elided arguments are filled in via the same rules used for pointer types. Here is an aggressive use of this feature:

```
typedef struct Lst<ρ1,ρ2> *ρ2 l_t<ρ1,ρ2>;
l_t heap_copy(l_t l) {
  l_t ans = NULL;
  for(l_t l2 = l; l2 != NULL; l2 = l2->tl)
    ans = new Lst(new *l2->hd,ans);
  return ans;
}
```

Because of defaults, the parameter type is `l_t<`ρ_1`,`ρ_2`>` and the return type is `l_t<`ρ_H`,`ρ_H`>`. Because of inference, the compiler gives `ans` the type `l_t<`ρ_H`,`ρ_H`>` (the `return` statement requires `ans` to have the function's return type) and `l2` the type `l_t<`ρ_1`,`ρ_2`>` (`l2`'s initializer has this type).

4.2 Interaction With Type Variables

Section 4.1.2 suggested that scope restrictions on region names prevent pointers from escaping the scope of their region. In particular, a function or block cannot return or assign a value of type $\tau*\rho$ outside the scope of ρ's definition, simply because you cannot write down a (well-formed) type for the result. Indeed, if Cyclone had no mechanism for type abstraction, this property would hold.

But if there is some way to hide a pointer's type in a result, then the pointer could escape the scope of its region. Existential types provide exactly this ability. (Closures and objects provide a similar ability in other languages, so the essential problem is first-class abstract types, which are crucial in safe strongly typed languages.) Hence Cyclone programs can create dangling pointers; safety demands that programs not dereference such pointers.

To address this problem, the type system keeps track of the set of region names that are considered live at each program point. Following Walker, Crary, and Morrisett [211], we call the set of live regions the *capability*. To allow dereferencing a pointer, the type system ensures that the associated region name is in the capability. Similarly, to allow a function call, Cyclone ensures that regions the function might access are all live. To this end, function types carry an *effect* that records the set of regions the function might access. The capability for a program point is the enclosed function's effect and the region names for all declaration blocks and dynamic-region statements containing the program point. The idea of using effects to ensure soundness is due to Tofte and Talpin [201]. However, Cyclone's effect system differs substantially from previous work.

Our first departure from Tofte and Talpin's system is that we calculate default effects from the function prototype alone (instead of inferring them from the function body) to preserve separate compilation. The default effect includes the set of region names that appear in the argument or result types. For instance, given the prototype:

```
int*ρ1 f(int*, int*ρ1*);
```

which elaborates to:

```
int*ρ1 f<ρ1,ρ2,ρ3>(int*ρ2, int*ρ1*ρ3);
```

the default effect is {ρ_1, ρ_2, ρ_3}.

In the absence of polymorphism, this default effect is a conservative bound on the regions the function might access. The programmer can override the default with an explicit effect. For example, if `f` never dereferences its first argument, we can strengthen its prototype by adding an explicit effect as follows:

```
int*ρ1 f(int*ρ2, int*ρ1*ρ3; {ρ1,ρ3});
```

Given this stronger type, callers could instantiate ρ_2 with the name of a (possibly) deallocated region, and therefore pass a dangling pointer. Unsurprisingly, using nondefault effects is exceedingly rare.

Our second departure from Tofte and Talpin's system is that we do not have *effect variables* (i.e., type variables with an effect kind). Effect variables serve three purposes: First, they simulate subtyping in a unification-based inference framework. Second, they abstract the set of regions a data-hiding construct might need to access. Third, they abstract the set of regions an abstract type hides.

Cyclone used effect variables at first, but we abandoned the approach for two reasons. First, to support effect subtyping correctly, the Tofte-Talpin inference algorithm requires that all effect variables are prenex quantified and each function

type has a unique effect variable in its effect [199]. Without these invariants, unification can fail. In an explicitly typed language like Cyclone, it is awkward to enforce these invariants. Furthermore, prenex quantification prevents first-class polymorphism, which Cyclone otherwise supports.

Second, effect variables appear in some library interfaces, making the libraries harder to understand and use. Consider a type for polymorphic sets (where ϵ is an effect variable):

```
struct Set<α,ρ,ϵ> {
  list_t<α,ρ> elts;
  int (*cmp)(α,α; ϵ);
};
```

A `Set` consists of a list of α elements, with the spine of the list in region ρ. We do not know where the elements are allocated until we instantiate α. The comparison function `cmp` determines set membership. Because the elements' type is not yet known, the type of `cmp` must use an effect variable ϵ to abstract the set of regions that it might access when comparing the two α values. This effect variable, like the type and region variable, must be abstracted by the `Set` structure.

Suppose the library exports `Set` to clients abstractly:

```
struct Set<α,ρ::R,ϵ::E>; // R for region kind, E for effect kind
```

The client must discern the connection between α and ϵ, namely that ϵ abstracts the set of regions within α that the hidden comparison function might access.

4.2.1 Avoiding Effect Variables

To simplify the system while retaining the benefit of effect variables, we use a type operator, regions(τ). This novel operator is just part of the type system; it does not exist at run time. Intuitively, regions(τ) represents the set of region names that occur free in τ. In particular:

$$\begin{aligned} \text{regions(int)} &= \emptyset \\ \text{regions}(\tau * \rho) &= \{\rho\} \cup \text{regions}(\tau) \\ \text{regions}(\tau\ (*\texttt{f})(\tau_1, \ldots, \tau_n)) &= \text{regions}(\tau) \cup \text{regions}(\tau_1) \cup \ldots \cup \text{regions}(\tau_n) \end{aligned}$$

For type variables, regions(α) is treated as an abstract set of region variables, much like an effect variable. For example, regions($\alpha * \rho$) = $\{\rho\} \cup$ regions(α). The default effect of a function that has α in its type simply includes regions(α). That way, when we instantiate α with τ, the resulting function type has an effect that includes the free region names in τ.

We can now rewrite the `Set` example as follows:

```
struct Set<α,ρ> {
  list_t<α,ρ> elts;
  int (*cmp)(α,α; regions(α));
};
```

Now the connection between the type parameter α and the function's effect is apparent, and the data structure no longer needs an effect-variable parameter. Moreover, regions(α) is the default effect for `int (*cmp)(`α`,`α`)`, so we need not write it.

Now suppose we wish to build a `Set<int*`ρ_1`,`ρ_2`>` value using a particular comparison function (with unnecessary annotations for expository purposes):

```
int cmp_ptr<ρ3,ρ4>(int*ρ3 p1, int*ρ4 p2) {
  return (*p1) == (*p2);
}
Set<int*ρ1,ρ2> build_set(list_t<int*ρ1,ρ2> e) {
  return Set{.elts = e, .cmp = cmp_ptr<ρ1,ρ1>};
}
```

The default effect for `cmp_ptr` is $\{\rho_1\}$. After instantiating α with `int*`ρ_1, the effect of `cmp` becomes `regions_of(int*`ρ_1`)`, which equals $\{\rho_1\}$. As a result, `build_set` type-checks. In fact, using any function with a default effect will always succeed. Consequently, programmers need not explicitly mention effects when designing or using libraries.

Our particular choice for the definition of regions(τ) is what ensures that programs with default effects and without dangling pointers never fail to type-check because of effects. (I do not prove this conjecture.) In essence, the definition is the most permissive for programs without dangling pointers, so it is a natural choice. Interestingly, any definition of regions(τ) that does not introduce type variables (i.e., regions(τ) must not include any type variable or region name not already free in τ) is sound. All that matters is that we substitute the same set for all occurrences of regions(α) so that we maintain any effect equalities that were assumed when type-checking the code for which α is in scope. For proof that any well-formed definition of regions(τ) is sound, observe that the proof in Appendix B uses no property of regions(τ) except that it does not introduce type variables.

4.2.2 Using Existential Types

As mentioned above, existential types allow Cyclone programs to create dangling pointers, as this example demonstrates:

```
struct IntFn {<α> int (*func)(α env); α env;};
int read<ρ>(int*ρ x) { return *x; }
struct IntFn dangle() {
  L:{int x = 0;
     struct IntFn ans =
        {<int*ρ_L> .func = read<ρ_L>, .env = &x};
     return ans; }
}
```

The witness type $\texttt{int*}\rho_L$ does not appear in the result type `struct IntFn`, so `dangle` is well-typed. Therefore, the type-checker rejects any attempted call to the `func` field of a `struct IntFn`:

```
int apply_intfn(struct IntFn pkg) {
  let IntFn{<β> .func = f,.env = y} = pkg;
  return f(y); // rejected
}
```

The effect of `f` is regions(β), but the pattern match does *not* add the bound type variables to the current capability because doing so is unsound. Every use of an existential package so far in this dissertation is ill-typed for this reason. To make existential packages usable in conjunction with the region system, we must "leak" enough information to prove a call is safe, without leaking so much information that we no longer hide data. Effect variables offer one solution. Instead, we enrich constraints, which we used above to indicate one region outlived another, to have the form $\epsilon_1 < \epsilon_2$ where ϵ_1 and ϵ_2 are effects. The constraint holds if for all variables α in ϵ_1 there exists a variable β in ϵ_2 such that α outlives β. For example, we can revise `struct IntFn` like this:

```
struct IntFn<ρ> {< α: α<ρ > int (*func)(α env); α env;};
```

The constraint defines a region bound: For any `struct IntFn`$<\rho>$, regions(α) outlive ρ, so having ρ in the current capability is sufficient to call `func`. For example, we can always use `struct IntFn`$<\rho_H>$, but the witness type cannot mention regions other than the heap. By allowing bounds other than ρ_H, we provide more flexibility than requiring all abstract types to live forever, but programmers unconcerned with memory management can just add a ρ_H bound for all existentially bound type variables. Doing so fixes our earlier examples.

4.3 Run-Time Support

The code-generation and run-time support for Cyclone regions is very simple. Heap and stack manipulation are exactly as in C. Dynamic regions are represented as

linked lists of "pages" where each page is twice the size of the previous one. A region handle points to the beginning of the list and the current "allocation point" on the last page, where `rnew` or `rmalloc` place the next object. If there is insufficient space for an object, a new page is allocated. Region deallocation frees each page.

When the garbage collector is included, dynamic-region list pages are acquired from the collector. The collector supports explicit deallocation, which we use to free regions. Note that the collector simply treats the region pages as large objects. They are always reachable from the stack, so they are scanned and any pointers to heap-allocated objects are found, ensuring that these objects are preserved. The advantage of this interface is its simplicity, but at some cost: At collection time, every object in every dynamic region appears reachable, and thus all (live) dynamic regions must be scanned, and no objects within (or reachable from) dynamic regions are reclaimed.

The code generator ensures that regions are deallocated even when their lifetimes end due to unstructured control flow. For each intraprocedural jump or `return`, it is easy to determine statically how many regions should be deallocated before transferring control. When throwing an exception, the number of regions to deallocate is not known statically. Therefore, we store region handles and exception handlers in an integrated list that operates in a last-in-first-out manner. When an exception is thrown, we traverse the list deallocating regions until we reach an exception handler. We then transfer control with `longjmp`. In this fashion, we ensure that a region is always deallocated when control returns.

4.4 Evaluation

This section informally evaluates the region system's strengths (the idioms it conveniently captures) and weaknesses (inconvenient restrictions and how we might lift them). The last section presents some advanced examples I encountered in practice and how the system supports them.

4.4.1 Good News

Cyclone's approach to memory management meets its primary goals. It preserves safety without resigning all addressable objects to a garbage-collected heap. There is no per-access run-time cost; the generated code for pointer dereferences is exactly the same as C. By grouping objects into regions, types of the form $\tau*\rho$ capture lifetime information in the type system without being so fine-grained that every pointer has a different type.

The lexically scoped lifetimes of Cyclone regions restricts coding idioms, as

described in Section 4.4.2, but it captures some of the most common idioms and contributes to eliminating explicit region annotations. C's local-declaration blocks already have lexically scoped lifetimes, so Cyclone's system describes them naturally. Functions that do not cause their parameters to "escape" (e.g., be stored in a data structure that outlives the function call) can always take the address of local variables. In C, passing the address of local variables is dangerous practice. In Cyclone, programmers are less hesitant to use this technique because the type system ensures it is safe.

Cyclone's dynamic regions capture the idiom where the caller determines a function result's lifetime but the callee determines the result's size. This division of responsibility is common: The result's size may depend on computation that only the callee should know about, but only the caller knows how the result will be used. In C, this idiom is more awkward to implement. If the callee allocates the result with `malloc`, the caller can use `free`, but it is difficult not to call `free` twice for the same memory. All too often, programs resort to a simpler interface in which the caller allocates space for the result that is "hopefully" large enough. The functions `gets` and `sprintf` in the C library are notorious examples. When the caller guesses wrong, the callee usually fails or commits a buffer overrun. Of course, C programs could implement dynamic regions.

Last-in-first-out lifetimes make Cyclone's region subtyping more useful: A region that is live on function entry always outlives regions that the function creates. We also need not worry about region-name aliasing. If a function could free a region named ρ before ρ left scope, then allowing access to a region named ρ' after the free is safe only if a caller cannot instantiate ρ and ρ' with the same region.

The integration with garbage collection lets programmers avoid the burden of manual memory management when their application does not need it. It also allows a convenient program-evolution path: Prototypes can rely on garbage collection and then use profiling to guide manual optimizations, such as using dynamic regions, to reduce memory consumption.

The default effects and region annotations work extremely well. Previously published work measured that it was possible to port some C applications to Cyclone by writing, on average, one explicit region annotation about every 200 lines [97]. A key to this result is that implicit instantiation of quantified types ensures callers write no extra information to use region-polymorphic functions.

Effects of the form regions(α) avoid effect variables for abstract container types. As a result, Cyclone programmers do not need to know about effects until they use existential types. Even then, simple region-bound constraints usually suffice.

The system actually has the full power of effect variables: If one uses regions(α) in an effect and α does not occur except in effects, then regions(α) imposes no more restrictions than an effect variable. However, inferring correct instantiations

for α is not guaranteed to succeed, so programs may need explicit instantiations. Nonetheless, this simulation of effect variables indicates that the Cyclone system, at least in its fully explicit form, is no less powerful due to its lack of effect variables.

4.4.2 Bad News

The biggest restriction we have imposed is that all regions have lexically scoped lifetimes. Hence garbage-collected heap objects are the only objects that can be reclaimed before their region leaves scope. We present some shortcomings of lexically scoped lifetimes before sketching an extension that safely allows programmers to deallocate regions at any program point and avoids per-access run-time cost. Greg Morrisett, my advisor, designed this extension, but its importance as a complement to more static regions warrants a brief description.

To understand the limits of lexical scope, consider the scheme of copying garbage collection couched in region-like terms: Create a region `r` and allocate all objects into it. When it becomes "too big," create a region `r2`, copy live data from `r` to `r2`, free `r`, and continue, using `r2` in place of `r`. With lexically scoped regions, we cannot reclaim `r` unless we create `r2` before `r`. But if we need to collect again, we must have already created an `r3`, and so on. Unless we can bound the number of garbage collections at compile-time, this scheme will not work. It is a common structure for long-running numerical calculations and event-based servers.

Another problem with lexical scope is that a global variable cannot point to nonheap memory unless the pointer is hidden by an existential type. (After all, ρ_H is the only region name with global scope.) If the pointer is hidden, the existential package cannot actually be used unless there is a region bound. But the bound would have to be ρ_H, which is true only for heap pointers. Hence garbage collection is the only way to reclaim memory accessible from global variables.

A third shortcoming is that a program's control structure can force regions to live longer than necessary. Here is an extreme example:

```
void f(int *x) {
  int y = *x;
  // ...run for a long time...
}
void g() {
  region r;
  // ...allocate a lot in r...
  int *p = rnew(r) 37;
  f(p);
}
```

It is safe to free the region that `g` creates as soon as the call to `f` initializes `y`.

To address these problems, we can add a new flavor of region that can be created and deallocated with an expression. Handles for such regions have types of the form `dynregion_t<`ρ`,`ρ'`>`, which means the region is named ρ and the handle is in ρ'. If the region named ρ' is deallocated, the region named ρ will also be deallocated, but ρ can be deallocated sooner. Primitive functions can allow creation and deallocation:

```
struct NewRegion<ρ'> {<ρ> dynregion_t<ρ,ρ'> d; };
struct NewRegion<ρ'> rnew_dynregion(region_t<ρ'>);
void free_dynregion(dynregion_t);
```

Because `rnew_dynregion` returns a new region, its name is existentially bound. As usual, unpacking the existential does not add ρ to the current capability. To use `dynregion_t<`ρ`,`ρ'`>`, we add the construct `region r = open` e`;` s where e has type `dynregion_t<`ρ`,`ρ'`>`. This construct throws an exception if e has been deallocated, else it binds `r` to a handle for the region, gives `r` the type `region_t<`ρ`>`, and adds ρ to the capability for s. Within s, any attempt to free the region (e.g., `free_dynregion(`e`)`) raises an exception. Hence we avoid any run-time cost for accessing objects in the region (in s), but opening and deallocating these regions require run-time checks and potential exceptions.

With this region flavor, we can mostly avoid the problems with lexical scope. However, for very long-running loops using the copying-collection technique, we still have the problem that the handles for the deallocated regions are not reclaimed. Support for restricted aliasing (of the handle) can avoid this shortcoming.

Turning to the type system, the rule that every pointer type has one region name can prove inconvenient. For example, consider this incorrect code:

```
int *ρ? f(bool b, int *ρ1 x, int *ρ2 y) {
  return b ? x : y;
}
```

No region name makes the return type correct because the function might return a pointer into the region named ρ_1 or a pointer into the region named ρ_2. We can use constraints to give this function a type:

```
int *ρ3 f(bool b, int *ρ1 x, int *ρ2 y: ρ1<ρ3, ρ2<ρ3) {
  return b ? x : y;
}
```

For data structures with pointers that could point into a variety of regions, we can use constraints on `struct` definitions in an analogous way. However, this technique

can be unnecessarily restrictive if part of the program does not know that can serve the purpose of ρ_3 in our example.

Another solution would annotate pointers with effects (including effect variables) instead of region names. For example, if ϵ_1 and ϵ_2 are effect variables, we could give `f` the type `int` $*\epsilon_1 \cup \epsilon_2$ `f(bool, int*`ϵ_1`, int*`ϵ_2`)`. To access $\tau * \epsilon$, the effect ϵ must be a subeffect of the current capability. The disadvantage of this extension is that it makes type inference more difficult. I believe it would require solving arbitrary abstract-set inequalities. For a language designed for humans, it is not clear that the added expressiveness justifies the added complications.

Another obvious limitation is that Cyclone programs cannot deallocate individual objects. Putting each object in its own region is not always an option. For example, one could not make a list of such objects because each list element would have a different type. Systems that restrict aliasing can allow such idioms. For example, if it is known that an acyclic list's elements are reachable only from the spine of the list, it is safe to deallocate the list's elements provided that the list is not used subsequently [212].

The region system suffers from some other less significant blemishes. First, the interface for dynamic regions is too coarse for resource-conscious programming. A wider interface could allow programs to set the initial page size, determine a policy for growing the region when the current page is full, set a maximum size for the region (beyond which allocation into it fails), and so on. Similarly,the interface to the garbage collector is so coarse that all objects in a dynamic region appear live and all fields of all objects are potentially pointers. More sophisticated interfaces are possible. For example, some regions could disallow pointers in them, so the collector would not need to scan them. Another possibility is setting a maximum object size (say n bytes) for a region and informing the collector. That way, a pointer to address p in such a region would cause the collector to scan only addresses $p - n$ to $p + n$.

Second, there is no way to keep a callee from allocating into the heap region, so the type system does little to prevent space leaks. Perhaps we should revisit the decision to make the heap region always accessible and outliving all other regions, but it is probably still the correct default for many applications.

Third, it is inconvenient to parameterize many `struct` definitions by the same region names. It is common to have a collection of interdependent (sometimes mutually recursive) type constructors where it suffices to parameterize all of them by a region name ρ and use ρ as the region name for all pointer types and type-constructor applications with the definitions.

On a different note, it is sound to allow subtyping on constraints, but the formalism in this chapter does not. For example, given two polymorphic functions

with different constraints, one may have a type that is a subtype of the other's type provided that its constraints imply the constraints of the other. Such subtyping amounts to bounded quantification over constraints. Chapter 3 referred to known results that bounded quantification over types makes subtyping undecidable. For constraints, the problem appears simpler because constraints just relate sets of type variables, but I have not carefully investigated decidability.

4.4.3 Advanced Examples

This section describes two sophisticated uses of existential types and their interaction with the region system. These examples help demonstrate Cyclone's power and the limitations of eliminating effect variables. They impose much more programmer burden than almost all other Cyclone code.

Closure Library The Cyclone closure library provides a collection of routines for manipulating closures (i.e., functions with hidden environments of abstract type), which we represent with this type:

```
struct Fn<α1,α2,ρ> { <α3> : regions(α3)<ρ
  α2 (*f)(α3,α1);
  α3 env;
};
typedef struct Fn<α1,α2,ρ> fn_t<α1,α2,ρ>;
```

The type `fn_t<`τ_0`,`τ_1`,`ρ`>` describes closures that produce a τ_1 given a τ_0. To call a closure's `f`, the capability must include regions(α_1), regions(α_2), and regions(α_3). The region bound means having ρ in a capability establishes regions(α_3). We can write routines to create and use closures:

```
fn_t<α1,α2,ρ> make_fn(α2 (*ρH f)(α3,α1), α3 x : regions(α3)<ρ) {
  return Fn{.f=f, .env=x};
}
α2 apply(fn_t<α1,α2> f, α1 x) {
  let Fn{<β> .f=code,.env=env} = f;
  return code(env,x);
}
```

In `apply`, the type-checker fills in the region bound for the type of `f`. Routines for other tasks, such as composing two closures or converting a function pointer to a closure, are also easy to write. More interesting are functions for currying and uncurrying. In languages in which all functions are closures, these functions have these types:

`curry` : $\forall\alpha_1,\alpha_2,\alpha_3.((\alpha_1 \times \alpha_2) \to \alpha_3) \to (\alpha_1 \to (\alpha_2 \to \alpha_3))$
`uncurry`: $\forall\alpha_1,\alpha_2,\alpha_3.(\alpha_1 \to (\alpha_2 \to \alpha_3)) \to ((\alpha_1 \times \alpha_2) \to \alpha_3)$

In Cyclone, we write `$(`τ_0`,`τ_1`)` for $\tau_0 \times \tau_1$, `$(`e_0`,`e_1`)` for tuple construction, and e`[i]` for tuple-field access. Implementing `uncurry` is straightforward:

```
α3 lambda(fn_t<α1,fn_t<α2,α3>> f, $(α1,α2)* arg) {
  return apply(apply(f,(*arg)[0]),(*arg)[1]);
}
fn_t<$(α1,α2)*ρ',α3,ρ> uncurry(fn_t<α1,fn_t<α2,α3,ρ>,ρ> f
                               : regions($(α1,α2,α3))<ρ) {
  return make_fn(lambda,f);
}
```

As usual, we build a closure that takes a pair and applies the two original closures appropriately. The explicit constraint in the type of `uncurry` seems redundant given the argument types, but Cyclone does not infer constraints from argument types. The only unnatural restriction is that the two original closures must have the same region bound. More lenient but even harder-to-read solutions exist: The two original closures could have bounds ρ_1 and ρ_2 and `uncurry` could have the additional constraints $\rho_1<\rho$ and $\rho_2<\rho$. Another possibility is to change the definition of `struct Fn` so that the bound is regions(β) for a type parameter β. Doing so simulates effect variables. Then `uncurry` could take closures with bounds β_1 and β_2 and return a closure with bound `$(`β_1`,`β_2`)`.

We can implement `curry` like this:

```
α3 inner($(fn_t<$(α1,α2)*,α3>,α1)* env, α2 second) {
  return apply((*env)[0],new $((*env)[1],second));
}
fn_t<α2,α3,ρ> outer(fn_t<$(α1,α2)*ρH,α3,ρ> f, α1 first
                    : regions($(α1,α2,α3))<ρ) {
  return make_fn(inner, new $(f,first));
}
fn_t<α1,fn_t<α2,α3,ρ>,ρ> curry(fn_t<$(α1,α2)*ρH,α3,ρ>
                               : regions($(α1,α2,α3))<ρ) {
  return make_fn(outer, f);
}
```

As usual, applying the first closure creates a second closure holding the first argument and the original closure in its environment. Applying the second closure applies the original closure to a newly created pair. The interesting point is that this solution type-checks only because the constraints for `outer` are discharged

when `outer` is instantiated in `curry`. Otherwise, the call to `make_fn` in `curry` would not type-check because its first parameter has a constraint-free function type. In fact, if the type-checker discharges constraints when a function is called, then it is impossible to implement `curry` given our definition of `struct Fn`.

Abstract Iterators Another example of an abstract data type is an iterator that returns successive elements from a hidden structure. We can define this type constructor:

```
struct Iter<α,ρ> {<β> : regions(β)<ρ
    β env;
    bool (*next)<ρ'>(β env, α*ρ' dest);
};
typedef struct Iter<α,ρ> iter_t<α,ρ>;
```

An iterator creator should provide a function for the `next` field that returns false when there are no more elements. When there is a next element, the function should store it in `*dest`. The existential type allows an iterator to maintain state to remember what elements remain. The first-class polymorphism for `next` (the universal quantification over ρ') allows each call to `next` to select where the next element is stored. For example, an iterator client could store some results on the stack and others in the heap. If ρ' were a parameter to `struct Iter`, all elements would have to be stored in one region (up to subtyping) and this region would have to be specified when creating the iterator.

The iterator library provides only one function:

```
bool next(iter_t<α> iter, α *dest) {
  let Iter{.env=env,.next=f} = iter;
  return f(env,dest);
}
```

The real work is in creating iterators. A representative example is an iterator for linked lists.

```
struct List<α,ρ> { α hd; struct List<α,ρ> *ρ tl; };
typedef struct List<α,ρ> *ρ list_t<α,ρ>;
bool iter_f<α,ρ1,ρ2,ρ3>(list_t<α,ρ1> *ρ2 elts_left, α *ρ3 dest) {
  if(!*elts_left)
    return false;
  *dest = (*elts_left)->hd;
  *elts_left = (*elts_left)->tl;
  return true;
}
iter_t<α,ρ> make_iter(region_t<ρ2> rgn, list_t<α,ρ1> lst
                      : regions(α)<ρ, {ρ1,ρ2}<ρ) {
  return Iter{.env=rnew(rgn) lst,
              .next = iter_f<α,ρ1,ρ2><>};
}
```

In `make_iter`, the witness type is `list_t<α,ρ1>*ρ2`: the private state is a pointer in ρ_2 to a list in ρ_1 with the remaining elements. To use `iter_f` for the `next` field we must *delay* the instantiation of its last type parameter, which is the purpose of the `<>` syntax. A minor point is that `iter_f` does not read the list elements, so we could give it an explicit effect omitting regions(α), which in turn should avoid needing the constraint regions(α)$<\rho$ in `make_iter`.

I have also implemented a more complicated iterator over red-black trees.

4.5 Formalism

This section defines a formal abstract machine that models most of the interesting aspects of the region system. In the machine, all objects are allocated into some region and regions obey a last-in-first-out discipline. Hence the run-time heap has more structure than a simple partial map (as in Chapter 3). We do not distinguish a heap region or model garbage collection. Type safety implies that programs do not access objects in deallocated regions. Furthermore, terminating programs deallocate all regions they allocate.

Although we have stack regions and dynamic regions, we do not prove that programs cannot allocate into stack regions after they are created. In fact, technically the static semantics allows handles for stack regions, but to prevent them it suffices to forbid explicit handles in source programs.

The type system is a combination of the type system for the formal machine in Chapter 3 and the additions necessary for modeling region types (type variables of kind `R`, effects on function types, constraints on quantified types, and singleton types for region handles). We provide subtyping for pointer types using the outlives

$$
\begin{array}{rrcl}
\text{kinds} & \kappa & ::= & \mathtt{B} \mid \mathtt{A} \mid \mathtt{R} \\
\text{effects} & \epsilon & ::= & \emptyset \mid \alpha \mid i \mid \epsilon \cup \epsilon \\
\text{constraints} & \gamma & ::= & \cdot \mid \gamma, \epsilon{<}\epsilon \\
\text{types} & \tau, r & ::= & \alpha \mid \text{int} \mid \tau \times \tau \mid \tau \xrightarrow{\epsilon} \tau \mid \tau{*}r \mid \forall\alpha{:}\kappa[\gamma].\tau \mid \exists\alpha{:}\kappa[\gamma].\tau \\
 & & \mid & \text{region}(r) \mid \text{S}(i) \\
\text{terms} & s & ::= & e \mid \mathsf{return}\ e \mid s; s \mid \mathsf{if}\ e\ s\ s \mid \mathsf{while}\ e\ s \mid \mathsf{let}\ \rho, x = e;\ s \\
 & & \mid & \mathsf{open}\ e\ \mathsf{as}\ \rho, \alpha, x;\ s \mid \mathsf{region}\ \rho, x\ s \mid s; \mathsf{pop}\ i \\
 & e & ::= & xp \mid i \mid f \mid \&e \mid {*}e \mid (e, e) \mid e.i \mid e{=}e \mid e(e) \mid \mathsf{call}\ s \\
 & & \mid & e[\tau] \mid \mathsf{pack}\ \tau, e\ \mathsf{as}\ \tau \mid \mathsf{rnew}\ e\ e \mid \mathsf{rgn}\ i \\
 & f & ::= & (\tau, \rho\ x) \xrightarrow{\epsilon} \tau\ s \mid \Lambda\alpha{:}\kappa[\gamma].f \\
 & p & ::= & \cdot \mid ip \\
\text{values} & v & ::= & i \mid \&xp \mid f \mid (v, v) \mid \mathsf{pack}\ \tau, v\ \mathsf{as}\ \tau \mid \mathsf{rgn}\ i \\
\text{heaps} & H & ::= & \cdot \mid H, x \mapsto v \\
 & S & ::= & \cdot \mid S, i{:}H \\
\text{states} & P & ::= & S; S; s \\
\text{contexts} & R & ::= & \cdot \mid R, i \\
 & \Delta & ::= & \cdot \mid \Delta, \alpha{:}\kappa \\
 & \Gamma & ::= & \cdot \mid \Gamma, x{:}(\tau, r) \\
 & C & ::= & R; \Delta; \Gamma; \gamma; \epsilon
\end{array}
$$

Figure 4.1: Chapter 4 Formal Syntax

relationship. For simplicity, we do not have subtyping on other types, though adding more subtyping would probably not be difficult.

The machine and proof are similar to those in an earlier technical report [98], but the version presented here makes some small improvements (and corrections) and is more like the formalisms in other chapters. In particular, the treatment of paths is like in Chapter 3, constraints allow $\epsilon_1{<}\epsilon_2$ instead of just $\epsilon{<}\rho$, and run-time regions are named with integers i instead of type variables ρ. This last difference is a matter of taste: It makes it clear that we do not use run-time type information, but it means we cannot use the type-variable context Δ to describe the regions that have been created. There is no difference for source programs. The formalism in Chapter 5 similarly uses integers for run-time locks instead of type variables.

4.5.1 Syntax

Figure 4.1 presents the language's syntax. Compared to the language in Chapter 3, we eliminate reference patterns and add the constructs necessary to reason about

Cyclone's region system. We focus on the additions.

Kinds include `R` for types that are region names. In source programs, only type variables α can have kind `R`. If we know α has kind `R`, we often write ρ to remind us, but α and ρ are in the same syntactic class. At run-time, we name actual regions with integers. If i names a region, then $\mathrm{S}(i)$ is a (singleton) type of kind `R`. If we know a type has kind `R`, we often write r instead of τ to remind us.

As in actual Cyclone, handles have types of the form region(r), and pointer types ($\tau * r$) include region names describing where the pointed-to value resides.

Function types include an explicit effect (ϵ) to describe regions that must be live before calling a function. In source programs, an effect is a set of type variables. For $\alpha \in \epsilon$, if α has kind `R`, we mean the region named α is live. More generally, for any α, we mean all region names mentioned in the type for which α stands are live. (It simplifies matters to allow α as an effect regardless of its kind.) At run-time, region names are integers, so effects include i. We assume effects are identical up to the usual notions of set equality (associativity, commutativity, idempotence). It is clear that our definition of substitution is identical for equal sets in this sense.

Quantified types can introduce constraints (γ). The constraint $\epsilon_1 < \epsilon_2$ means that if ϵ_2 describes only live regions, then ϵ_1 describes only live regions. Put another way, assuming ϵ_2 is live, we can conclude ϵ_1 is live. To instantiate a universal type, the constraints must hold after instantiation. Similarly, for an existential package to have an existential type, the type's constraints must hold given the package's witness type. The constraint $\epsilon_1 < \epsilon_2$ holds if for all type variables $\alpha \in \epsilon_1$, there exists a type variable $\beta \in \epsilon_2$ such that $\alpha < \beta$. The last-in-first-out nature of regions introduces constraints: When allocating a region named ρ, we know $\epsilon < \rho$ where ϵ is the current capability.

Most term forms are similar to terms in Chapter 3. The let and open constructs allocate one location in a "stack region" (a region that is deallocated after the enclosed statement terminates and for which there is no handle). The compile-time region name for this region is the ρ following the variable name; ρ is bound in the enclosed statement. The term $\mathsf{region}\ \rho, x\ s$ corresponds to `region r` s in Cyclone: It creates a region, binds its handle to x (placing x in the region itself), executes s, and deallocates the region. The explicit ρ is like ρ_r in Cyclone. The statement form $s; \mathsf{pop}\ i$ does not appear in source programs. It is a placeholder in the term syntax so that the machine deallocates region i after executing s.

The expression forms are from Chapter 3 except $\mathsf{rnew}\ e_1\ e_2$ and $\mathsf{rgn}\ i$. The former is exactly like `rnew(`e_1`)` e_2 in Cyclone. The latter is an actual region handle for region i. Region handles are values. Function definitions include explicit effects and constraints to keep type-checking syntax-directed, and an explicit region name for the parameter (instead of inducing a region name from the function name).

The heap has structure corresponding to regions. S is a stack of regions, each

of which maps locations to values, with the most recently allocated region on the right. We assume the regions i in S are distinct and that their domains are unique (i.e., a variable in one H is not repeated in any other H). By abuse of notation, we write $x \notin S_1 S_2$ to mean there is no H in S_1 or S_2 such that $x \in \mathrm{Dom}(H)$.

A program state $S_G; S; s$ includes *garbage* data S_G, *live* data S, and current code s. The machine becomes stuck if it tries to access S_G, so S_G does not effect program behavior. It contains the deallocated regions that, in practice, we would not keep at run-time. An explicit S_G is just a technical device to keep program states from referring to free variables, even if they have dangling pointers.

To type-check terms, we use a context (C) to specify the run-time region names in scope (R), the kinds of type variables (Δ), the types and regions of locations (Γ), the known constraints (γ), and the current capability (ϵ). For source programs, R is empty or perhaps contains a predefined "heap region." Given program state $S_G; S; s$, we use S_G and S to induce an R and a γ. Section 4.5.3 presents the details of how the heap affects the typing context. As convenient, given $C = R; \Delta; \Gamma; \gamma; \epsilon$, we write C_R, C_Δ, C_Γ, C_γ, and C_ϵ for R, Δ, Γ, γ, and ϵ, respectively.

When juxtaposing two partial maps (e.g., $\Gamma_1\Gamma_2$), we mean their union and implicitly require that their domains are disjoint. Similarly, $R_1 R_2$ means R_1 followed by R_2. We do *not* implicitly consider an R reorderable. In particular, the $\vdash_{\text{spop}}$ and $\vdash_{\text{epop}}$ judgments use R to restrict the order that a program deallocates regions. When order is unimportant, we may treat R as a set, writing $i \in R$ to mean R has the form R_1, i, R_2 and $R \subseteq R'$ to mean that if $i \in R$ then $i \in R'$.

4.5.2 Dynamic Semantics

As in Chapter 3, the rules for rewriting P to P' are defined in terms of interdependent judgments for statements, left-expressions, and right-expressions (Figures 4.2); accessing and mutating parts of aggregate objects are defined using auxiliary judgments (Figure 4.4); and type instantiation involves type substitution (Figure 4.5), which has no essential run-time effect. We now describe these judgments in more detail.

Rule DS4.1 creates a new region to hold the (stack) object v. It puts the region to the right of S because it will be deallocated before the regions in S. The region's run-time name is some fresh i, so we substitute $\mathrm{S}(i)$ for ρ in s. If we used type variables for run-time region names, we could rely on α-conversion to ensure ρ was fresh and avoid type substitution. To deallocate i at the right time, we insert the appropriate pop statement. Rules DS4.2–7 are just like rules DS3.2–7 in Chapter 3. Rule DS4.8 creates a new dynamic region. It is just like DS4.1 except x holds a handle (rgn i) for the new region. Rules DS4.9 and DS4.10 are elimination rules for pop; they deallocate regions. They apply only if the region i is the rightmost live

$$\frac{x \notin S_G S \quad i \notin \mathrm{Dom}(S_G S)}{S_G; S; \mathsf{let}\ \rho, x = v;\ s \xrightarrow{\mathrm{s}} S_G; S, i{:}x \mapsto v; (s; \mathsf{pop}\ i)[\mathrm{S}(i)/\rho]}\ \text{DS4.1}$$

$$\frac{}{S_G; S; (v; s) \xrightarrow{\mathrm{s}} S_G; S; s}\ \text{DS4.2} \qquad \frac{}{S_G; S; (\mathsf{return}\ v; s) \xrightarrow{\mathrm{s}} S_G; S; \mathsf{return}\ v}\ \text{DS4.3}$$

$$\frac{}{S_G; S; \mathsf{if}\ 0\ s_1\ s_2 \xrightarrow{\mathrm{s}} S_G; S; s_2}\ \text{DS4.4} \qquad \frac{i \neq 0}{S_G; S; \mathsf{if}\ i\ s_1\ s_2 \xrightarrow{\mathrm{s}} S_G; S; s_1}\ \text{DS4.5}$$

$$\frac{}{S_G; S; \mathsf{while}\ e\ s \xrightarrow{\mathrm{s}} S_G; S; \mathsf{if}\ e\ (s; \mathsf{while}\ e\ s)\ 0}\ \text{DS4.6}$$

$$\frac{}{S_G; S; \mathsf{open}(\mathsf{pack}\ \tau', v\ \mathsf{as}\ \exists\alpha{:}\kappa[\gamma].\tau)\ \mathsf{as}\ \rho, \alpha, x;\ s \xrightarrow{\mathrm{s}} S_G; S; \mathsf{let}\ \rho, x = v;\ s[\tau'/\alpha]}\ \text{DS4.7}$$

$$\frac{x \notin S_G S \quad i \notin \mathrm{Dom}(S_G S)}{S_G; S; \mathsf{region}\ \rho, x\ s \xrightarrow{\mathrm{s}} S_G; S, i{:}x \mapsto \mathsf{rgn}\ i; (s; \mathsf{pop}\ i)[\mathrm{S}(i)/\rho]}\ \text{DS4.8}$$

$$\frac{}{S_G; S, i{:}H; (v; \mathsf{pop}\ i) \xrightarrow{\mathrm{s}} S_G, i{:}H; S; v}\ \text{DS4.9}$$

$$\frac{}{S_G; S, i{:}H; (\mathsf{return}\ v; \mathsf{pop}\ i) \xrightarrow{\mathrm{s}} S_G, i{:}H; S; \mathsf{return}\ v}\ \text{DS4.10}$$

$$\frac{S_G; S; e \xrightarrow{\mathrm{r}} S'_G; S'; e'}{\begin{array}{r} S_G; S; e \xrightarrow{\mathrm{s}} S'_G; S'; e' \\ S_G; S; \mathsf{return}\ e \xrightarrow{\mathrm{s}} S'_G; S'; \mathsf{return}\ e' \\ S_G; S; \mathsf{if}\ e\ s_1\ s_2 \xrightarrow{\mathrm{s}} S'_G; S'; \mathsf{if}\ e'\ s_1\ s_2 \\ S_G; S; \mathsf{let}\ \rho, x = e;\ s \xrightarrow{\mathrm{s}} S'_G; S'; \mathsf{let}\ \rho, x = e';\ s \\ S_G; S; \mathsf{open}\ e\ \mathsf{as}\ \rho, \alpha, x;\ s \xrightarrow{\mathrm{s}} S'_G; S'; \mathsf{open}\ e'\ \mathsf{as}\ \rho, \alpha, x;\ s \end{array}}\ \text{DS4.11}$$

$$\frac{S_G; S; s \xrightarrow{\mathrm{s}} S'_G; S'; s'}{\begin{array}{r} S_G; S; (s; s_2) \xrightarrow{\mathrm{s}} S'_G; S'; (s'; s_2) \\ S_G; S; (s; \mathsf{pop}\ i) \xrightarrow{\mathrm{s}} S'_G; S'; (s'; \mathsf{pop}\ i) \end{array}}\ \text{DS4.12}$$

Figure 4.2: Chapter 4 Dynamic Semantics, Statements

$$\frac{\mathrm{get}(H(x), p, v)}{S_G; S, i{:}H, S'; xp \xrightarrow{\mathrm{r}} S_G; S, i{:}H, S'; v}\ \text{DR4.1}$$

$$\frac{\mathrm{set}(v', p, v, v'')}{S_G; S, i{:}H, x \mapsto v', H', S'; xp{=}v \xrightarrow{\mathrm{r}} S_G; S, i{:}H, x \mapsto v'', H', S'; v}\ \text{DR4.2}$$

$$\frac{}{S_G; S; *\&xp \xrightarrow{\mathrm{r}} S_G; S; xp}\ \text{DR4.3} \qquad \frac{}{S_G; S; (v_0, v_1).i \xrightarrow{\mathrm{r}} S_G; S; v_i}\ \text{DR4.4}$$

$$\frac{}{S_G; S; ((\tau, \rho\ x) \xrightarrow{\epsilon} \tau'\ s)(v) \xrightarrow{\mathrm{r}} S_G; S; \mathsf{call}\ (\mathsf{let}\ \rho, x = v;\ s)}\ \text{DR4.5}$$

$$\frac{}{S_G; S; \mathsf{call\ return}\ v \xrightarrow{\mathrm{r}} S_G; S; v}\ \text{DR4.6} \qquad \frac{}{S_G; S; (\Lambda\alpha{:}\kappa[\gamma].f)[\tau] \xrightarrow{\mathrm{r}} S_G; S; f[\tau/\alpha]}\ \text{DR4.7}$$

$$\frac{x \notin S_G S S' \quad x \notin \mathrm{Dom}(H)}{S_G; S, i{:}H, S'; \mathsf{rnew\ rgn}\ i\ v \xrightarrow{\mathrm{r}} S_G; S, i{:}H, x \mapsto v, S'; \&x\cdot}\ \text{DR4.8}$$

$$\frac{S_G; S; s \xrightarrow{\mathrm{s}} S'_G; S'; s'}{S_G; S; \mathsf{call}\ s \xrightarrow{\mathrm{r}} S'_G; S'; \mathsf{call}\ s'}\ \text{DR4.9} \qquad \frac{S_G; S; e \xrightarrow{\mathrm{l}} S'_G; S'; e'}{\begin{array}{c} S_G; S; \&e \xrightarrow{\mathrm{r}} S'_G; S'; \&e' \\ S_G; S; e{=}e_2 \xrightarrow{\mathrm{r}} S'_G; S'; e'{=}e_2 \end{array}}\ \text{DR4.10}$$

$$\frac{S_G; S; e \xrightarrow{\mathrm{r}} S'_G; S'; e'}{\begin{array}{rr} & S_G; S; (e, e_2) \xrightarrow{\mathrm{r}} S'_G; S'; (e', e_2) \\ S_G; S; *e \xrightarrow{\mathrm{r}} S'_G; S'; *e' & S_G; S; (v, e) \xrightarrow{\mathrm{r}} S'_G; S'; (v, e') \\ S_G; S; e.i \xrightarrow{\mathrm{r}} S'_G; S'; e'.i & S_G; S; e(e_2) \xrightarrow{\mathrm{r}} S'_G; S'; e'(e_2) \\ S_G; S; xp{=}e \xrightarrow{\mathrm{r}} S'_G; S'; xp{=}e' & S_G; S; v(e) \xrightarrow{\mathrm{r}} S'_G; S'; v(e') \\ S_G; S; e[\tau] \xrightarrow{\mathrm{r}} S'_G; S'; e'[\tau] & S_G; S; \mathsf{rnew}\ e\ e_2 \xrightarrow{\mathrm{r}} S'_G; S'; \mathsf{rnew}\ e'\ e_2 \\ & S_G; S; \mathsf{rnew}\ v\ e \xrightarrow{\mathrm{r}} S'_G; S'; \mathsf{rnew}\ v\ e' \\ \multicolumn{2}{c}{S_G; S; \mathsf{pack}\ \tau', e\ \mathsf{as}\ \exists\alpha{:}\kappa[\gamma].\tau \xrightarrow{\mathrm{r}} S'_G; S'; \mathsf{pack}\ \tau', e'\ \mathsf{as}\ \exists\alpha{:}\kappa[\gamma].\tau} \end{array}}\ \text{DR4.11}$$

$$\frac{}{S_G; S; (xp).i \xrightarrow{\mathrm{l}} S_G; S; xpi}\ \text{DL4.1} \qquad \frac{}{S_G; S; *\&xp \xrightarrow{\mathrm{l}} S_G; S; xp}\ \text{DL4.2}$$

$$\frac{S_G; S; e \xrightarrow{\mathrm{r}} S'_G; S'; e'}{S_G; S; *e \xrightarrow{\mathrm{l}} S'_G; S; *e'}\ \text{DL4.3} \qquad \frac{S_G; S; e \xrightarrow{\mathrm{l}} S'_G; S'; e'}{S_G; S; e.i \xrightarrow{\mathrm{l}} S'_G; S'; e'.i}\ \text{DL4.4}$$

Figure 4.3: Chapter 4 Dynamic Semantics, Expressions

$$\frac{}{\mathrm{get}(v, \cdot, v)} \qquad \frac{\mathrm{get}(v_0, p, v)}{\mathrm{get}((v_0, v_1), 0p, v)} \qquad \frac{\mathrm{get}(v_1, p, v)}{\mathrm{get}((v_0, v_1), 1p, v)}$$

$$\frac{}{\mathrm{set}(v', \cdot, v, v)} \qquad \frac{\mathrm{set}(v_0, p, v, v')}{\mathrm{set}((v_0, v_1), 0p, v, (v', v_1))} \qquad \frac{\mathrm{set}(v_1, p, v, v')}{\mathrm{set}((v_0, v_1), 1p, v, (v_0, v'))}$$

Figure 4.4: Chapter 4 Dynamic Semantics, Heap Objects

region, else the machine is stuck. The rules add the region to the garbage stack. The region's position in the garbage stack is irrelevant because this stack is never accessed. The congruence rules DS4.11–12 hold no surprises. As in Chapter 3, putting multiple conclusions in one rule is just for conciseness.

Rules DR4.1 and DR4.2 use the get and set relations (which are simpler than in Chapter 3 because we eliminated reference patterns) to access or update the live data. They are complicated only because of the extra structure in S. Most importantly, they do not use S_G; the machine is stuck if the active term is xp and $x \in S_G$. All of the other rules are analogues of rules in Chapter 3, except for DR4.8. This rule defines allocation into a dynamic region. It creates a new location x, puts v in it, and returns a pointer to x.

Unlike Cyclone, regions(τ) is not a syntactic effect. Rather, effects are the union of "primitive effects," which have the form α or i. This decision simplifies the static judgments regarding effects and constraints, but it slightly complicates the definition of type substitution through effects: For effect α, we define $\alpha[\tau/\alpha]$ to be regions(τ), where regions is a metafunction from types to effects defined in Figure 4.5. The type-safety proof uses the fact that regions(τ) produces an effect that is well-formed so long as τ is well-formed. The rest of the definition of substitution is conventional.

4.5.3 Static Semantics

A valid source program is a statement s that type-checks under an empty context $(\cdot; \cdot; \cdot; \cdot; \emptyset; \tau \vdash_{\mathrm{styp}} s)$, does not terminate without returning $(\vdash_{\mathrm{ret}} s)$, and does not contain any pop statements $(\cdot \vdash_{\mathrm{spop}} s)$. As in Chapter 3, the type-checking judgments for statements, left-expressions, and right-expressions (Figures 4.8 and 4.9) are interdependent and the gettype relation (Figure 4.10) destructs the types of aggregate objects. The most interesting change is that type-checking a left-expression determines its type and the region name describing its location.

Expressions that access memory (e.g., assignment and the right-expression xp)

$$
\begin{array}{lrcl}
\text{effect substitution:} & \emptyset[\tau/\alpha] & = & \emptyset \\
 & \alpha[\tau/\alpha] & = & \text{regions}(\tau) \\
 & \beta[\tau/\alpha] & = & \beta \\
 & i[\tau/\alpha] & = & i \\
 & (\epsilon_1 \cup \epsilon_2)[\tau/\alpha] & = & (\epsilon_1[\tau/\alpha]) \cup (\epsilon_2[\tau/\alpha]) \\
\text{constraint substitution:} & \cdot[\tau/\alpha] & = & \cdot \\
 & (\gamma, \epsilon_1{<}\epsilon_2)[\tau/\alpha] & = & \gamma[\tau/\alpha], \epsilon_1[\tau/\alpha]{<}\epsilon_2[\tau/\alpha] \\
\text{type substitution:} & \alpha[\tau/\alpha] & = & \tau \\
 & \beta[\tau/\alpha] & = & \beta \\
 & \text{int}[\tau/\alpha] & = & \text{int} \\
 & (\tau_0 \times \tau_1)[\tau/\alpha] & = & \tau_0[\tau/\alpha] \times \tau_1[\tau/\alpha] \\
 & (\tau_1 \xrightarrow{\epsilon} \tau_2)[\tau/\alpha] & = & \tau_1[\tau/\alpha] \xrightarrow{\epsilon[\tau/\alpha]} \tau_2[\tau/\alpha] \\
 & (\tau' * r)[\tau/\alpha] & = & (\tau'[\tau/\alpha]) * (r[\tau/\alpha]) \\
 & (\forall\beta{:}\kappa[\gamma].\tau')[\tau/\alpha] & = & \forall\beta{:}\kappa[\gamma[\tau/\alpha]].\tau'[\tau/\alpha] \\
 & (\exists\beta{:}\kappa[\gamma].\tau')[\tau/\alpha] & = & \exists\beta{:}\kappa[\gamma[\tau/\alpha]].\tau'[\tau/\alpha] \\
 & (\text{region}(r))[\tau/\alpha] & = & \text{region}(r[\tau/\alpha]) \\
 & \text{S}(i)[\tau/\alpha] & = & \text{S}(i) \\
 & & & \\
\text{constraint regions:} & \text{regions}(\cdot) & = & \emptyset \\
 & \text{regions}(\gamma, \epsilon_1{<}\epsilon_2) & = & \text{regions}(\gamma) \cup \epsilon_1 \cup \epsilon_2 \\
\text{type regions:} & \text{regions}(\alpha) & = & \alpha \\
 & \text{regions}(\text{int}) & = & \emptyset \\
 & \text{regions}(\tau_0 \times \tau_1) & = & \text{regions}(\tau_0) \cup \text{regions}(\tau_1) \\
 & \text{regions}(\tau_1 \xrightarrow{\epsilon} \tau_2) & = & \epsilon \\
 & \text{regions}(\tau * r) & = & \text{regions}(\tau) \cup \text{regions}(r) \\
 & \text{regions}(\forall\alpha{:}\kappa[\gamma].\tau) & = & (\text{regions}(\gamma) \cup \text{regions}(\tau)) - \alpha \\
 & \text{regions}(\exists\alpha{:}\kappa[\gamma].\tau) & = & (\text{regions}(\gamma) \cup \text{regions}(\tau)) - \alpha \\
 & \text{regions}(\text{region}(r)) & = & \text{regions}(r) \\
 & \text{regions}(\text{S}(i)) & = & i
\end{array}
$$

Notes: Throughout, we mean $\beta \neq \alpha$ and implicitly rename to avoid capture. We omit the formal definition for terms and contexts; we simply substitute through all contained terms, types, constraints, and effects.

Figure 4.5: Chapter 4 Dynamic Semantics, Type Substitution

$$\frac{}{R;\Delta \vdash_{\text{wf}} \emptyset} \qquad \frac{\alpha \in \text{Dom}(\Delta)}{R;\Delta \vdash_{\text{wf}} \alpha} \qquad \frac{i \in R}{R;\Delta \vdash_{\text{wf}} i} \qquad \frac{R;\Delta \vdash_{\text{wf}} \epsilon_1 \quad R;\Delta \vdash_{\text{wf}} \epsilon_2}{R;\Delta \vdash_{\text{wf}} \epsilon_1 \cup \epsilon_2}$$

$$\frac{}{R;\Delta \vdash_{\text{wf}} \cdot} \qquad \frac{R;\Delta \vdash_{\text{wf}} \gamma \quad R;\Delta \vdash_{\text{wf}} \epsilon_1 \quad R;\Delta \vdash_{\text{wf}} \epsilon_2}{R;\Delta \vdash_{\text{wf}} \gamma, \epsilon_1 < \epsilon_2}$$

$$\frac{}{R;\Delta \vdash_{\text{k}} \text{int} : \texttt{B}} \qquad \frac{\alpha \in \text{Dom}(\Delta)}{R;\Delta \vdash_{\text{k}} \alpha : \kappa} \qquad \frac{i \in R}{R;\Delta \vdash_{\text{k}} S(i) : \texttt{R}} \qquad \frac{R;\Delta \vdash_{\text{k}} \tau : \texttt{B}}{R;\Delta \vdash_{\text{k}} \tau : \texttt{A}}$$

$$\frac{R;\Delta \vdash_{\text{k}} \tau_0 : \texttt{A} \quad R;\Delta \vdash_{\text{k}} \tau_1 : \texttt{A}}{R;\Delta \vdash_{\text{k}} \tau_0 \times \tau_1 : \texttt{A}} \qquad \frac{R;\Delta \vdash_{\text{k}} \tau_0 : \texttt{A} \quad R;\Delta \vdash_{\text{k}} \tau_1 : \texttt{A} \quad R;\Delta \vdash_{\text{wf}} \epsilon}{R;\Delta \vdash_{\text{k}} \tau_0 \xrightarrow{\epsilon} \tau_1 : \texttt{A}}$$

$$\frac{R;\Delta \vdash_{\text{k}} \tau : \texttt{A} \quad R;\Delta \vdash_{\text{k}} r : \texttt{R}}{R;\Delta \vdash_{\text{k}} \tau * r : \texttt{B}} \qquad \frac{R;\Delta \vdash_{\text{k}} r : \texttt{R}}{R;\Delta \vdash_{\text{k}} \text{region}(r) : \texttt{B}}$$

$$\frac{R;\Delta, \alpha{:}\kappa \vdash_{\text{k}} \tau : \texttt{A} \quad R;\Delta, \alpha{:}\kappa \vdash_{\text{wf}} \gamma \quad \kappa \neq \texttt{A} \quad \alpha \notin \text{Dom}(\Delta)}{\begin{array}{c} R;\Delta \vdash_{\text{k}} \forall \alpha{:}\kappa[\gamma].\tau : \texttt{A} \\ R;\Delta \vdash_{\text{k}} \exists \alpha{:}\kappa[\gamma].\tau : \texttt{A} \end{array}}$$

$$\frac{}{R;\Delta \vdash_{\text{wf}} \cdot} \qquad \frac{R;\Delta \vdash_{\text{wf}} \Gamma \quad R;\Delta \vdash_{\text{k}} \tau : \texttt{A} \quad R;\Delta \vdash_{\text{k}} r : \texttt{R}}{R;\Delta \vdash_{\text{wf}} \Gamma, x{:}(\tau, r)}$$

$$\frac{R;\Delta \vdash_{\text{wf}} \Gamma \quad R;\Delta \vdash_{\text{wf}} \gamma \quad R;\Delta \vdash_{\text{wf}} \epsilon}{\vdash_{\text{wf}} R;\Delta;\Gamma;\gamma;\epsilon}$$

Figure 4.6: Chapter 4 Kinding and Well-Formedness

type-check only if the current capability and constraints establish that the memory has not been deallocated. The judgment $\gamma;\epsilon \vdash_{\text{acc}} r$ defines this notion using the more general notion of subeffecting that $\gamma \vdash_{\text{eff}} \epsilon_1 < \epsilon_2$ defines. We also use the latter judgment to type-check a function call because the current capability must establish the function's effect. Finally, we need to lift this notion of implication to constraints ($\gamma \vdash_{\text{eff}} \gamma'$) to check the introduction of quantified types with constraints. Figure 4.7 defines these judgments.

Figure 4.6 defines the judgments for ensuring types have the correct kinds and typing contexts are well-formed. Constraints, effects, and types may not have free occurrences of type variables and regions not in the R and Δ provided as context.

The judgments in Figures 4.12 and 4.13 are more sophisticated than is necessary for source programs. The judgments $R \vdash_{\text{spop}} s$ and $R \vdash_{\text{epop}} e$ relax the requirement that programs have no pop statements while still imposing enough structure to ensure that programs deallocate regions in the correct order and do not access

$$\frac{\gamma \vdash_{\text{eff}} i \Leftarrow \epsilon}{\gamma; \epsilon \vdash_{\text{acc}} \mathrm{S}(i)} \qquad \frac{\gamma \vdash_{\text{eff}} \rho \Leftarrow \epsilon}{\gamma; \epsilon \vdash_{\text{acc}} \rho}$$

$$\frac{}{\gamma \vdash_{\text{eff}} \epsilon \Leftarrow \epsilon} \qquad \frac{}{\gamma_1, \epsilon_1 < \epsilon_2, \gamma_2 \vdash_{\text{eff}} \epsilon_1 \Leftarrow \epsilon_2}$$

$$\frac{\gamma \vdash_{\text{eff}} \epsilon_1 \Leftarrow \epsilon \quad \gamma \vdash_{\text{eff}} \epsilon_2 \Leftarrow \epsilon}{\gamma \vdash_{\text{eff}} \epsilon_1 \cup \epsilon_2 \Leftarrow \epsilon} \qquad \frac{\gamma \vdash_{\text{eff}} \epsilon \Leftarrow \epsilon_1}{\gamma \vdash_{\text{eff}} \epsilon \Leftarrow \epsilon_1 \cup \epsilon_2} \qquad \frac{\gamma \vdash_{\text{eff}} \epsilon_1 \Leftarrow \epsilon_3 \quad \gamma \vdash_{\text{eff}} \epsilon_3 \Leftarrow \epsilon_2}{\gamma \vdash_{\text{eff}} \epsilon_1 \Leftarrow \epsilon_2}$$

$$\frac{}{\gamma \vdash_{\text{eff}} .} \qquad \frac{\gamma \vdash_{\text{eff}} \gamma' \quad \gamma \vdash_{\text{eff}} \epsilon_1 \Leftarrow \epsilon_2}{\gamma \vdash_{\text{eff}} \gamma', \epsilon_1 < \epsilon_2}$$

Figure 4.7: Chapter 4 Effect and Constraint Containment

$$\frac{C \vdash_{\text{rtyp}} e : \tau'}{C; \tau \vdash_{\text{styp}} e} \text{ SS4.1} \qquad \frac{C \vdash_{\text{rtyp}} e : \tau}{C; \tau \vdash_{\text{styp}} \mathsf{return}\ e} \text{ SS4.2} \qquad \frac{C; \tau \vdash_{\text{styp}} s_1 \quad C; \tau \vdash_{\text{styp}} s_2}{C; \tau \vdash_{\text{styp}} s_1; s_2} \text{ SS4.3}$$

$$\frac{C \vdash_{\text{rtyp}} e : \text{int} \quad C; \tau \vdash_{\text{styp}} s}{C; \tau \vdash_{\text{styp}} \mathsf{while}\ e\ s} \text{ SS4.4} \qquad \frac{C \vdash_{\text{rtyp}} e : \text{int} \quad C; \tau \vdash_{\text{styp}} s_1 \quad C; \tau \vdash_{\text{styp}} s_2}{C; \tau \vdash_{\text{styp}} \mathsf{if}\ e\ s_1\ s_2} \text{ SS4.5}$$

$$\frac{\begin{array}{c} R; \Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} e : \tau' \qquad \rho \notin \mathrm{Dom}(\Delta) \qquad x \notin \mathrm{Dom}(\Gamma) \\ R; \Delta, \rho{:}\mathtt{R}; \Gamma, x{:}(\tau', \rho); \gamma, \epsilon{<}\rho; \epsilon \cup \rho; \tau \vdash_{\text{styp}} s \qquad R; \Delta \vdash_{\text{k}} \tau : \mathtt{A} \end{array}}{R; \Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{\text{styp}} \mathsf{let}\ \rho, x = e;\ s} \text{ SS4.6}$$

$$\frac{\begin{array}{c} R; \Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} e : \exists \alpha{:}\kappa[\gamma'].\tau' \qquad \alpha, \rho \notin \mathrm{Dom}(\Delta) \qquad x \notin \mathrm{Dom}(\Gamma) \\ R; \Delta, \rho{:}\mathtt{R}, \alpha{:}\kappa; \Gamma, x{:}(\tau', \rho); \gamma, \epsilon{<}\rho, \gamma'; \epsilon \cup \rho; \tau \vdash_{\text{styp}} s \qquad R; \Delta \vdash_{\text{k}} \tau : \mathtt{A} \end{array}}{R; \Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{\text{styp}} \mathsf{open}\ e\ \mathsf{as}\ \rho, \alpha, x;\ s} \text{ SS4.7}$$

$$\frac{\begin{array}{c} \vdash_{\text{wf}} R; \Delta; \Gamma; \gamma; \epsilon \qquad \rho \notin \mathrm{Dom}(\Delta) \qquad x \notin \mathrm{Dom}(\Gamma) \\ R; \Delta, \rho{:}\mathtt{R}; \Gamma, x{:}(\mathrm{region}(\rho), \rho); \gamma, \epsilon{<}\rho; \epsilon \cup \rho; \tau \vdash_{\text{styp}} s \qquad R; \Delta \vdash_{\text{k}} \tau : \mathtt{A} \end{array}}{R; \Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{\text{styp}} \mathsf{region}\ \rho, x\ s} \text{ SS4.8}$$

$$\frac{R; \Delta; \Gamma; \gamma; \epsilon \cup i; \tau \vdash_{\text{styp}} s}{R; \Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{\text{styp}} s; \mathsf{pop}\ i} \text{ SS4.9}$$

Figure 4.8: Chapter 4 Typing, Statements

$$\frac{C_\Gamma(x) = (\tau', r) \quad \vdash \text{gettype}(\tau', p, \tau) \quad \vdash_{\text{wf}} C}{C \vdash_{\text{ltyp}} xp : \tau, r} \text{ SL4.1}$$

$$\frac{C \vdash_{\text{rtyp}} e : \tau{*}r}{C \vdash_{\text{ltyp}} {*}e : \tau, r} \text{ SL4.2} \qquad \frac{C \vdash_{\text{ltyp}} e : \tau_0 \times \tau_1, r}{C \vdash_{\text{ltyp}} e.0 : \tau_0, r} \text{ SL4.3} \qquad \frac{C \vdash_{\text{ltyp}} e : \tau_0 \times \tau_1, r}{C \vdash_{\text{ltyp}} e.1 : \tau_1, r} \text{ SL4.4}$$

$$\frac{C \vdash_{\text{ltyp}} e : \tau, r' \quad C_\gamma; \text{regions}(r) \vdash_{\text{acc}} r' \quad C_R; C_\Delta \vdash_{\text{k}} r : \mathtt{R}}{C \vdash_{\text{ltyp}} e : \tau, r} \text{ SL4.5}$$

$$\frac{C_\Gamma(x) = (\tau', r) \quad \vdash \text{gettype}(\tau', p, \tau) \quad C_\gamma; C_\epsilon \vdash_{\text{acc}} r \quad \vdash_{\text{wf}} C}{C \vdash_{\text{rtyp}} xp : \tau} \text{ SR4.1}$$

$$\frac{C \vdash_{\text{rtyp}} e : \tau{*}r \quad C_\gamma; C_\epsilon \vdash_{\text{acc}} r}{C \vdash_{\text{rtyp}} {*}e : \tau} \text{ SR4.2} \quad \frac{C \vdash_{\text{rtyp}} e : \tau_0 \times \tau_1}{C \vdash_{\text{rtyp}} e.0 : \tau_0} \text{ SR4.3} \quad \frac{C \vdash_{\text{rtyp}} e : \tau_0 \times \tau_1}{C \vdash_{\text{rtyp}} e.1 : \tau_1} \text{ SR4.4}$$

$$\frac{\vdash_{\text{wf}} C}{C \vdash_{\text{rtyp}} i : \text{int}} \text{ SR4.5} \qquad \frac{C \vdash_{\text{ltyp}} e : \tau, r}{C \vdash_{\text{rtyp}} \&e : \tau{*}r} \text{ SR4.6} \qquad \frac{C \vdash_{\text{rtyp}} e_0 : \tau_0 \quad C \vdash_{\text{rtyp}} e_1 : \tau_1}{C \vdash_{\text{rtyp}} (e_0, e_1) : \tau_0 \times \tau_1} \text{ SR4.7}$$

$$\frac{C \vdash_{\text{ltyp}} e_1 : \tau, r \quad C \vdash_{\text{rtyp}} e_2 : \tau \quad C_\gamma; C_\epsilon \vdash_{\text{acc}} r}{C \vdash_{\text{rtyp}} e_1{=}e_2 : \tau} \text{ SR4.8}$$

$$\frac{C \vdash_{\text{rtyp}} e_1 : \tau' \xrightarrow{\epsilon'} \tau \quad C \vdash_{\text{rtyp}} e_2 : \tau' \quad C_\gamma \vdash_{\text{eff}} \epsilon' \Leftarrow C_\epsilon}{C \vdash_{\text{rtyp}} e_1(e_2) : \tau} \text{ SR4.9} \qquad \frac{C; \tau \vdash_{\text{styp}} s \quad \vdash_{\text{ret}} s}{C \vdash_{\text{rtyp}} \mathsf{call}\ s : \tau} \text{ SR4.10}$$

$$\frac{C \vdash_{\text{rtyp}} e : \forall \alpha{:}\kappa[\gamma'].\tau' \quad C_R; C_\Delta \vdash_{\text{k}} \tau : \kappa \quad C_\gamma \vdash_{\text{eff}} \gamma'[\tau/\alpha]}{C \vdash_{\text{rtyp}} e[\tau] : \tau'[\tau/\alpha]} \text{ SR4.11}$$

$$\frac{C \vdash_{\text{rtyp}} e : \tau[\tau'/\alpha] \quad C_R; C_\Delta \vdash_{\text{k}} \tau' : \kappa \quad C_\gamma \vdash_{\text{eff}} \gamma'[\tau'/\alpha] \quad C_R; C_\Delta \vdash_{\text{k}} \exists \alpha{:}\kappa[\gamma'].\tau : \mathtt{A}}{C \vdash_{\text{rtyp}} \mathsf{pack}\ \tau', e\ \mathsf{as}\ \exists \alpha{:}\kappa[\gamma'].\tau : \exists \alpha{:}\kappa[\gamma'].\tau} \text{ SR4.12}$$

$$\frac{\begin{array}{l}\vdash_{\text{wf}} R; \Delta; \Gamma; \gamma; \epsilon \qquad \rho \notin \text{Dom}(\Delta) \qquad x \notin \text{Dom}(\Gamma) \\ R; \Delta, \rho{:}\mathtt{R}; \Gamma, x{:}(\tau, \rho); \gamma, \epsilon'{<}\rho; \epsilon' \cup \rho; \tau' \vdash_{\text{styp}} s \qquad \vdash_{\text{ret}} s \qquad R; \Delta \vdash_{\text{k}} \tau \xrightarrow{\epsilon'} \tau' : \mathtt{A}\end{array}}{R; \Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} (\tau, \rho\ x) \xrightarrow{\epsilon'} \tau'\ s : \tau \xrightarrow{\epsilon'} \tau'} \text{ SR4.13}$$

$$\frac{R; \Delta, \alpha{:}\kappa; \Gamma; \gamma\gamma'; \epsilon \vdash_{\text{rtyp}} f : \tau \quad \vdash_{\text{wf}} R; \Delta; \Gamma; \gamma; \epsilon \quad R; \Delta \vdash_{\text{k}} \forall \alpha{:}\kappa[\gamma'].\tau : \mathtt{A}}{R; \Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} \Lambda \alpha{:}\kappa[\gamma'].f : \forall \alpha{:}\kappa[\gamma'].\tau} \text{ SR4.14}$$

$$\frac{\begin{array}{l}C \vdash_{\text{rtyp}} e_2 : \tau \\ C \vdash_{\text{rtyp}} e_1 : \text{region}(r) \quad C_\gamma; C_\epsilon \vdash_{\text{acc}} r\end{array}}{C \vdash_{\text{rtyp}} \mathsf{rnew}\ e_1\ e_2 : \tau{*}r} \text{ SR4.15} \qquad \frac{i \in C_R \quad \vdash_{\text{wf}} C}{C \vdash_{\text{rtyp}} \mathsf{rgn}\ i : \text{region}(S(i))} \text{ SR4.16}$$

$$\frac{C \vdash_{\text{rtyp}} e : \tau{*}r' \quad C_\gamma; \text{regions}(r) \vdash_{\text{acc}} r' \quad C_R; C_\Delta \vdash_{\text{k}} r : \mathtt{R}}{C \vdash_{\text{rtyp}} e : \tau{*}r} \text{ SR4.17}$$

Figure 4.9: Chapter 4 Typing, Expressions

$$\frac{}{\vdash \text{gettype}(\tau, \cdot, \tau)} \qquad \frac{\vdash \text{gettype}(\tau_0, p, \tau)}{\vdash \text{gettype}(\tau_0 \times \tau_1, 0p, \tau)} \qquad \frac{\vdash \text{gettype}(\tau_1, p, \tau)}{\vdash \text{gettype}(\tau_0 \times \tau_1, 1p, \tau)}$$

Figure 4.10: Chapter 4 Typing, Heap Objects

$$\frac{}{\vdash_{\text{ret}} \mathsf{return}\ e} \qquad \frac{\vdash_{\text{ret}} s_1 \quad \vdash_{\text{ret}} s_2}{\vdash_{\text{ret}} \mathsf{if}\ e\ s_1\ s_2} \qquad \frac{\vdash_{\text{ret}} s}{\begin{array}{ll} \vdash_{\text{ret}} s; s' & \vdash_{\text{ret}} \mathsf{let}\ \rho, x = e;\ s \\ \vdash_{\text{ret}} s'; s & \vdash_{\text{ret}} \mathsf{open}\ e\ \mathsf{as}\ \rho, \alpha, x;\ s \\ \vdash_{\text{ret}} s; \mathsf{pop}\ i & \vdash_{\text{ret}} \mathsf{region}\ \rho, x\ s \end{array}}$$

Figure 4.11: Chapter 4 Must-Return

$$\frac{R \vdash_{\text{spop}} s}{i, R \vdash_{\text{spop}} s; \mathsf{pop}\ i} \qquad \frac{R \vdash_{\text{spop}} s_1 \quad \cdot \vdash_{\text{spop}} s_2}{R \vdash_{\text{spop}} s_1; s_2} \qquad \frac{R \vdash_{\text{epop}} e \quad \cdot \vdash_{\text{spop}} s_1 \quad \cdot \vdash_{\text{spop}} s_2}{R \vdash_{\text{spop}} \mathsf{if}\ e\ s_1\ s_2} \qquad \frac{\cdot \vdash_{\text{epop}} e \quad \cdot \vdash_{\text{epop}} s}{\cdot \vdash_{\text{spop}} \mathsf{while}\ e\ s}$$

$$\frac{R \vdash_{\text{epop}} e \quad \cdot \vdash_{\text{spop}} s}{\begin{array}{l} R \vdash_{\text{spop}} \mathsf{let}\ \rho, x = e;\ s \\ R \vdash_{\text{spop}} \mathsf{open}\ e\ \mathsf{as}\ \rho, \alpha, x;\ s \end{array}} \qquad \frac{\cdot \vdash_{\text{spop}} s}{\cdot \vdash_{\text{spop}} \mathsf{region}\ \rho, x\ s} \qquad \frac{R \vdash_{\text{epop}} e}{\begin{array}{l} R \vdash_{\text{spop}} e \\ R \vdash_{\text{spop}} \mathsf{return}\ e \end{array}}$$

$$\frac{}{\begin{array}{l} \cdot \vdash_{\text{epop}} xp \\ \cdot \vdash_{\text{epop}} i \\ \cdot \vdash_{\text{epop}} \mathsf{rgn}\ i \end{array}} \qquad \frac{R \vdash_{\text{epop}} e}{\begin{array}{l} R \vdash_{\text{epop}} \&e \\ R \vdash_{\text{epop}} *e \\ R \vdash_{\text{epop}} e.i \\ R \vdash_{\text{epop}} xp{=}e \\ R \vdash_{\text{epop}} e[\tau] \\ R \vdash_{\text{epop}} \mathsf{pack}\ \tau, e\ \mathsf{as}\ \tau' \end{array}} \qquad \frac{R \vdash_{\text{epop}} e_0 \quad \cdot \vdash_{\text{epop}} e_1}{\begin{array}{l} R \vdash_{\text{epop}} (e_0, e_1) \\ R \vdash_{\text{epop}} e_0{=}e_1 \\ R \vdash_{\text{epop}} e_0(e_1) \\ R \vdash_{\text{epop}} \mathsf{rnew}\ e_0\ e_1 \end{array}} \qquad \frac{\cdot \vdash_{\text{epop}} v \quad R \vdash_{\text{epop}} e}{\begin{array}{l} R \vdash_{\text{epop}} (v, e) \\ R \vdash_{\text{epop}} v(e) \\ R \vdash_{\text{epop}} \mathsf{rnew}\ v\ e \end{array}}$$

$$\frac{\cdot \vdash_{\text{spop}} s}{\cdot \vdash_{\text{epop}} (\tau, \rho\ x) \xrightarrow{\epsilon} \tau'\ s} \qquad \frac{\cdot \vdash_{\text{epop}} f}{\cdot \vdash_{\text{epop}} \Lambda\alpha{:}\kappa[\gamma].f} \qquad \frac{R \vdash_{\text{spop}} s}{R \vdash_{\text{epop}} \mathsf{call}\ s}$$

Figure 4.12: Chapter 4 Typing, Deallocation

$$\frac{}{R;\Gamma;\gamma;i \vdash_{\text{htyp}} \cdot : \cdot} \qquad \frac{R;\Gamma;\gamma;i \vdash_{\text{htyp}} H : \Gamma' \quad R;\cdot;\Gamma;\gamma;\emptyset \vdash_{\text{rtyp}} v : \tau \quad \cdot \vdash_{\text{epop}} v}{R;\Gamma;\gamma;i \vdash_{\text{htyp}} H, x \mapsto v : \Gamma', x{:}(\tau, S(i))}$$

$$\frac{}{R;\Gamma;\gamma \vdash_{\text{htyp}} \cdot : \cdot} \qquad \frac{R;\Gamma;\gamma \vdash_{\text{htyp}} S : \Gamma_1 \quad R;\Gamma;\gamma;i \vdash_{\text{htyp}} H : \Gamma_2}{R;\Gamma;\gamma \vdash_{\text{htyp}} S, i{:}H : \Gamma_1\Gamma_2}$$

$$\frac{\begin{array}{l} S = i_1{:}H_1, \ldots, i_n{:}H_n \qquad R = i_1, \ldots, i_n \qquad \gamma = i_1{<}i_2, i_2{<}i_3, \ldots, i_{n-1}{<}i_n \\ S_G = i'_1{:}, H'_1 \ldots, i'_m{:}H'_m \quad R_G = i'_1, \ldots, i'_m \quad \gamma_G = \epsilon_1{<}i'_1, \ldots, \epsilon_m{<}i'_m \\ R_G R;\Gamma;\gamma\gamma_G \vdash_{\text{htyp}} S_G S : \Gamma \qquad R_G R;\cdot;\Gamma;\gamma\gamma_G;\emptyset;\tau \vdash_{\text{styp}} s \qquad \vdash_{\text{ret}} s \qquad R \vdash_{\text{spop}} s \end{array}}{\vdash_{\text{prog}} S_G; S; s}$$

Figure 4.13: Chapter 4 Typing, States

deallocated regions. The two $\vdash_{\text{htyp}}$ judgments derive a context Γ for an S. Finally, the judgment $\vdash_{\text{prog}}$ type-checks entire program states.

We now describe the judgments in more detail, omitting descriptions of the more straightforward type-checking rules.

Except for $R;\Delta \vdash_{\text{k}} \tau : \kappa$, the judgments in Figure 4.6 just ensure that constraints and effects do not refer to meaningless free type variables or regions. The kinding judgment is simpler than in Chapter 3 because all types are mutable and have known size. To ensure the latter, the rule for quantified types forbids a type variable of kind `A`. The most interesting aspect of the judgment is the use of kind `R`: The rules require that r in $\tau{*}r$, region(r), and (τ, r) (where $\Gamma(x) = (\tau, r)$) should have kind `R`. No other types should have kind `R`.

The $\vdash_{\text{acc}}$ and $\vdash_{\text{eff}}$ judgments are mostly straightforward. Given implicit set equalities such as associativity, commutativity, and $\epsilon = \epsilon \cup \emptyset$, the rules for $\gamma \vdash_{\text{eff}} \epsilon_1{<}\epsilon_2$ amount to showing that given the constraints γ, for all α in ϵ_1 there exists a β in ϵ_2 such that α outlives β.

Rules SS4.6–8 are complicated because they concern terms that create regions. For each, to type-check the contained statement s, we extend Δ with the compile-time name for the created region (ρ), extend Γ with x having the appropriate type and region-name ρ, extend the current capability with ϵ (because the region is live while s executes), and extend γ with $\epsilon{<}\rho$ (because the region lifetimes are last-in-first-out). If our language did not have composite operations (allocate a region, allocate an object into it, execute a statement, deallocate the region), we would not have such complicated rules. The other hypotheses in these rules are straightforward. Note that in SS4.8 the handle has type region(ρ). Rule SS4.9 just adds i to the current capability to check the s in $s;\mathsf{pop}\ i$. It is important that we not include i in other capabilities (e.g., to check s' in $(s;\mathsf{pop}\ i);s'$) even though

i is live because doing so would make it impossible for the type-safety proof to establish that s' could still type-check after i was deallocated.

The rules for $C \vdash_{\text{ltyp}} e : \tau, r$ are quite simple. The region name for $e.i$ is the same as the region name for e because an aggregate object resides in one region. None of the rules use the $\vdash_{\text{acc}}$ judgment explicitly because evaluation of left-expressions does not access memory unless evaluation of a contained right-expression does so. Not requiring left-expressions to refer to live memory allows dangling pointers (e.g., $\&xp$) to type-check even if the pointed-to memory has been deallocated.

On the other hand, the rules for type-checking right expressions do use the $\vdash_{\text{acc}}$ and $\vdash_{\text{eff}}$ judgments. For example, if SR4.1 did not have its $\vdash_{\text{acc}}$ hypothesis, then we could not prove that the expression xp was not stuck. (As we will see, C_ϵ will not include deallocated regions.) Rule SR4.6 is one reason $\vdash_{\text{ltyp}}$ includes a region name; we need it for the type of $\&e$. The other reason is rule SR4.8, where we need the region name to forbid mutating deallocated memory. Rule SR4.9 forbids function calls unless the current capability establishes the function's effect. Rules SR4.10 and SR4.11 ensure that constraints introduced with quantified types are provable from the known constraints in the context. Constraints never "become false," so these rules make it sound to assume a quantified type's constraints in rules SR4.14 and SS4.7. Rules SL4.5 and SR4.17 allow subtyping of pointer types. (Earlier work [98] erroneously omitted SL4.5. The language is safe without it, but type preservation does not hold.)

The $\vdash_{\text{ret}}$ judgment, defined in Figure 4.11, is used for rules SR4.10 and SR4.13, much like in Chapter 3.

The intuition behind $R \vdash_{\text{spop}} s$ is that s should deallocate the regions in R in right-to-left order and deallocate no region twice. (Because $s; \mathsf{pop}\ i$ executes s before deallocating i, it is correct that i should be the left-most region in R.) Furthermore, if s terminates, it should deallocate all the region in R. The actual definition is slightly more restrictive. For example, it requires all of the pop statements in s to be nested inside each other. More technically, the abstract-syntax path from the root of s to the active redex must include all pop statements.

The $\vdash_{\text{htyp}}$ judgments add $x{:}(\tau, S(i))$ to Γ if region i maps x to a value v of type τ. Values never need a nonempty capability to type-check (they do not execute), so we can type-check v with $\emptyset$ for ϵ. We must require $\cdot \vdash_{\text{epop}} v$ to ensure that function bodies in the heap do not have pop statements.

Finally, there is one rule for $\vdash_{\text{prog}} S_G; S; s$. As usual, the heap must type-check (allowing cyclic references) and s must type-check under the heap's context. There should be no free occurrences of type variables ($\Delta = \cdot$) and only regions in S_G or S should be used. We type-check the heap and s using the constraints γ and γ_G. It is sound to assume γ because $R \vdash_{\text{spop}} s$ ensures s will deallocate the regions in S in the order consistent with γ. As for γ_G, given $i \in R_G$, it is sound to use any

constraint of the form $\epsilon{<}i$. This constraint means, "if i describes a live region, then ϵ describes only live regions," which holds vacuously because i is not live.

4.5.4 Type Safety

Appendix B proves this result:

Definition 4.1. *State $S_G; S; s$ is stuck if s is not of the form* return v *and there are no S'_G, S', and s' such that $S_G; S; s \xrightarrow{s} S'_G; S'; s'$.*

Theorem 4.2 (Type Safety). *If $\cdot;\cdot;\cdot;\cdot;\emptyset;\tau \vdash_{\text{styp}} s$, $\vdash_{\text{ret}} s$, s contains no pop statements, and $\cdot;\cdot;s \xrightarrow{s}{}^{*} S'_G; S'; s'$ (where $\xrightarrow{s}{}^{*}$ is the reflexive, transitive closure of $\xrightarrow{s}$), then $S'_G; S'; s'$ is not stuck.*

Note that any "attempt" to access an object in the garbage heap (S_G) would lead to a stuck state.

4.6 Related Work

As a safe polymorphic C-like language with static region-based memory management, Cyclone represents a unique and promising point in the programming-language design-space, but many other systems share some of its features. This section describes some of these systems' approaches to memory management.

Making C Safe Many systems aim to make C code safe, as Chapter 8 describes. Some static bug-finding tools, such as LCLint and its successor Splint [63, 189], perform unsound but useful analyses to find potential dangling-pointer dereferences and space leaks. Annotations can describe invariants such as reference counting and pointer uniqueness. Like Cyclone, avoiding analysis errors requires restricted coding idioms or additional annotations, but unlike Cyclone, soundness is not guaranteed. In this way, static tools reduce false positives without rewriting code.

Other systems, such as Safe-C [12], change data representation and insert run-time checks to detect dangling-pointer dereferences at run-time. For example, C pointers can be compiled to machine addresses plus integers representing the object pointed to and code manipulating pointers can maintain the integers as well. The run-time system maintains a table of "live" integers and dereferences require checking the table. (Integers are never reused.) Performance overhead is substantial. To allow stack pointers, activation records also need associated integers. Because this approach requires changing data representation, it becomes difficult to link against legacy object code. Therefore, some systems keep the integers in a separate table indexed by the pointer values [129].

The CCured system [164, 38] takes a hybrid approach to recover most of the performance. When an object is freed, the (entire) storage is not immediately reclaimed, but rather marked as inaccessible. Subsequent accesses check the mark and signal an error when the object is dereferenced. Ultimately, the mark is reclaimed with a garbage collector to avoid leaks. A whole-program static analysis ensures that dangling stack pointers do not exist. When the analysis is too conservative, programmers must rewrite their code.

The main advantage of all these systems is that they require less modification of legacy C code. However, none soundly preserve data representation and object lifetimes, which are common reasons for using C.

Static Regions Tofte and Talpin's seminal work [201] on implementing ML with regions provides the foundation for regions in the ML Kit [200]. Programming with the Kit is convenient, as the compiler automatically infers all region annotations. However, small changes to a program can have drastic, unintuitive effects on object lifetimes. Thus, to program effectively, one must understand the analysis and try to control it indirectly by using certain idioms [200]. More recent work for the ML Kit includes optional support for accurate garbage collection within regions [103]. Doing so requires changing region inference so that it never creates dangling pointers.

A number of extensions to the basic Tofte-Talpin framework can avoid the constraints of last-in-first-out region lifetimes. As examples, the ML Kit includes a reset-region primitive [200] (Cyclone has experimented with this feature); Aiken et al. provide an analysis to free some regions early [3]; and Walker et al. [210, 211, 213] propose general systems for freeing regions based on linear types. These systems are more expressive than our framework. For instance, the ideas in the Capability Calculus were used to implement type-safe garbage collectors *within* a language [214, 153]. However, these systems were not designed for source-level programming. They were designed as compiler intermediate languages or analyses, so they can ignore issues such as minimizing annotations or providing control to the user.

Two other recent projects, Vault [55] and the work of Henglein et al. [115] aim to provide safe source-level control over memory management using regions. Vault's powerful type system allows a region to be freed before it leaves scope and its types can enforce that code *must* free a region. To do so, Vault restricts region aliasing and tracks more fine-grained effects. As a result, programming in Vault can require more annotations. Henglein et al. [115] have designed a flexible region system that does not require last-in-first-out behavior. However, the system is monomorphic and first-order; it is unclear how to extend it to support

polymorphism or existential types, the key difficulties in this chapter.

Finally, both Typed Assembly Language [156] and the Microsoft CIL [91] provide some support for type-safe stack allocation. But neither system allows programmers to mix stack and heap pointers, and both systems place strong restrictions on how stack pointers can be used. For instance, the Microsoft CIL prevents such pointers from being placed in data structures or returned as results.

Regions in C Perhaps the most closely related work is Gay and Aiken's RC [85, 84] compiler and their earlier system, C@ [86]. They provide language support for efficient reference counting to detect if a region is deallocated while there remain pointers to it (that are not within it). This dynamic system has no *a priori* restrictions on regions' lifetimes and a pointer can point anywhere, so the RC approach can encode more memory-management idioms. RC is less eager in that it does not detect errors at compile-time, but more eager in that it fails when dangling references exist, rather than when they are followed. RC is not a safe language, but its approach to regions is sound.

Three pointer qualifiers keep RC's overhead low by imposing invariants that make reference-counting unnecessary. In general, the invariants are checked at run-time, but static analysis removes most checks. First, `traditional` pointers always point into the heap or stack. Because RC does not include these areas in its region system, using `traditional` pointers involves no reference-counting. Second, `sameregion` pointers always point into the region of the containing object. Because reference counts track only pointers from *outside* the region, RC can again avoid reference counting. Cyclone uses region-name equalities to capture the same-region idiom; without reference counting, the fact that a pointer points into its container's region is unimportant at run time. Third, RC's region-creation construct can take a "parent" region. The run-time checks that the parent region is freed after the new region. The `parentptr` qualifier is like `sameregion` except it allows pointers into ancestor regions. Parent pointers are like Cyclone's region subtyping.

Reference counting forces two restrictions not in Cyclone. First, RC forbids casts from `int` to `void*` (in Cyclone terms, instantiating α with `int`) because it leads to code that does not know if it is manipulating a pointer into a region. Cyclone's region system does not need to know this information, nor does the conservative garbage collector. Second, RC forbids `longjmp` (or in Cyclone terms, exceptions) because code that decrements reference counts due to local variables would not be executed. (Finalizers for activation records could avoid this problem.)

Other Regions Because the general idea of region-based memory management (allocating object into regions for which all objects are deallocated simultaneously)

is an old one, it is not possible to document all its uses. Gay and Aiken [86] nicely summarize many systems that use regions, including many that do not require simultaneous deallocation. Regions are sometimes called arenas [105] or zones.

In some sense, optimizing compilers that use analyses to stack-allocate objects are related. Essentially, Cyclone provides programmers this technique and the type system verifies that it is used soundly.

Because most garbage collectors are inappropriate for real-time tasks, the Real-Time Specification for Java [20] extends Java with "`ScopedMemory`" objects, which are essentially regions. In Cyclone terms, the creation of such an object (essentially a handle) is separate from a lexically scoped use of the region. The default location for allocated objects is the most recently used `ScopedMemory` object (or the heap if none are in use). Users can allocate objects elsewhere explicitly or set a new default. As in Cyclone, this scheme creates an implicit stack of regions and a region's objects are deallocated when control leaves the appropriate scope.

Unlike in Cyclone, the lifetime of a Real-Time Java object is not part of its type. Instead, attempting to create a reference from an older region to a younger one causes a run-time exception. Hence every assignment statement must include this lifetime check (though static analysis could eliminate some checks) and dangling pointers never exist. It is also incorrect for a `ScopedMemory` object to occur twice in a region stack; an exception occurs at the second attempted use. This error is impossible in Cyclone because we do not separate the creation of handles from the use of regions.

In summary, some systems are more convenient to use than Cyclone (e.g., CCured and the MLKit) but take away control over memory management. Some of the static systems (e.g., the Capability Calculus) provide more powerful region constructs, but were designed as intermediate languages and do not have the programming convenience of Cyclone. Other systems (e.g., RC, Safe-C) are more flexible but offer no static guarantees.

Chapter 5

Type-Safe Multithreading

This chapter extends Cyclone with locks and threads. Programs can create, acquire, and release locks, as well as spawn new threads. Threads communicate via shared mutable heap locations. To enforce safety, we extend the type system to enforce mutual exclusion on shared data, but we allow unsynchronized access to thread-local data. The extensions interact smoothly with the parametric polymorphism and region-based memory management that preceding chapters develop. To begin, we motivate safe multithreading and mutual exclusion. We then sketch this chapter's structure and highlight the technical contributions.

Writing multithreaded programs is more difficult than writing single-threaded programs because there are typically far too many possible execution sequences for human reasoning or even automated testing. In particular, it is easy for an unintended data race—one thread accessing data while another thread mutates the data—to leave program data in an inconsistent state. Because there are many multithreaded applications where C-style data representation and resource management are important (e.g., operating systems), extending Cyclone with multithreading makes it useful for important application domains.

Programmers often intend for their programs not to have race conditions. For this reason alone, extending the type system to guarantee that data races cannot occur is useful. It eliminates a potential source of errors and allows a thread to violate an invariant temporarily (e.g., to update a shared data structure) with the assurance that other threads cannot view data in a state violating the invariant.

In fact, preventing data races in multithreaded Cyclone is essential: In the presence of such races, Cyclone is not type-safe. The Cyclone implementation does not ensure that reads and writes of words in memory are atomic. After all, the system just uses a conventional C compiler (`gcc`) and a native thread library. If the underlying architecture, such as a shared-memory multiprocessor, does not prevent data races from corrupting memory, then a race condition on a pointer

could yield an arbitrary result, which of course violates memory safety.

Moreover, a system enforcing atomic access for words is insufficient because safety can require writing multiple words without an intervening access. When mutating an existential package such that its witness type changes, it is unsafe to allow access while some fields use the old witness type and some use the new. A common example is a `struct` with a field holding the length of an array that another field points to. Allowing updates of such records (to refer to shorter or longer arrays) is desirable, but we must forbid access while the length field is wrong.

In short, we have three reasons to enrich Cyclone's type system to guarantee the absence of data races:

1. Most programs are not supposed to have races, so static assurances increase reliability.

2. Updating references may not be atomic in the implementation, so races might corrupt pointers.

3. Type safety can require writes to multiple memory locations before another thread reads any of them.

These reasons should apply in some form to any expressive, safe, low-level, multithreaded language. From the perspective of designing a type-safe language, the first is "optional," but the others are "mandatory."

Section 5.1 describes Cyclone's basic techniques for making potential data races a compile-time error. The approach is strikingly similar to the approach for making dangling-pointer dereferences a compile-time error. We have compile-time *lock names* for run-time locks. Each lock type and pointer type includes a lock name. Two lock types with the same lock name describe the same run-time lock. A pointer type's lock name indicates a lock that mediates access to the pointed-to data. The type system ensures a thread accesses data only if it holds the appropriate lock.

The crucial complication is a notion of *thread-local data*. Such memory does not need a lock, but the type system must enforce that only one thread uses the memory. Thread-local data is often the rule, not the exception. Such data makes programs easier to write and more efficient. The kind system distinguishes sharable and unsharable types, but one library can let clients pass thread-local or shared data to it.

Section 5.2 describes the interaction between type variables representing locks and type variables representing types. As in Chapter 4, we need a way to describe the access rights necessary for using a value of an unknown type. As before, we use a novel type constructor and compile-time constraints. Unlike with last-in-first-out regions, we do not have a natural notion of subtyping.

Section 5.3 describes the interaction between multithreading and the region system. (Earlier sections ignore deallocation.) Mostly the systems are analogous but orthogonal. The interesting interaction comes from allowing threads to share data that does not live forever. We must prevent one thread from accessing data that another has deallocated.

Section 5.4 describes the necessary run-time support for multithreading. Because Cyclone's multithreading operations are quite conventional, it is easy to implement them on top of a native thread system. However, the interaction with regions requires some interesting run-time data structures.

Section 5.5 evaluates the system. The main strengths are uniformity with the region system and efficient access to shared data structures. The main weakness is the lack of support for synchronization idioms besides lock-based mutual exclusion.

Sections 5.6 and Appendix C model many interesting aspects of multithreaded Cyclone and prove a type-safety result. Because the abstract machine requires mutation to take two steps, type safety implies the absence of data races. This semantics models the difficulty of ensuring safety in the presence of nonatomic operations, but it significantly complicates the safety proof. To regain some simplicity, the model omits memory deallocation and left-expressions of the form e`.i`.

Finally, Section 5.7 describes related work. This chapter largely adapts closely related work on race-detection type systems for higher-level languages. In particular, Flanagan, Abadi, and Freund developed the idea of using singleton lock-types and effects [73, 72, 74]. They also applied their ideas to large Java programs. Furthermore, Boyapati, Lee, and Rinard's approach to thread-local data [31, 29] is very similar to mine. Nonetheless, this chapter makes the following technical contributions beyond adapting others' ideas:

- We integrate parametric polymorphism, which complicates the effect language as with regions. The result works particularly well for "caller locks" idioms.

- Callers can pass a special "`nonlock`" with thread-local data to callees that use a "callee locks" idiom. This addition allows more code reuse than Boyapati et al.'s system while incurring essentially no unnecessary overhead in the thread-local case.

- The integration with regions allows shared data objects that do not live forever.

- The kind system collects the above additions into a coherent type language that clearly describes what types are sharable.

- The type-safety proof is the first for a formal machine with thread-local data. Furthermore, previous formal work has prevented data races only for abstract machines in which races cannot actually violate type safety.

5.1 Basic Constructs

In this section, we present the extensions for Cyclone multithreading. Our design goals include the following:

- Statically enforce mutual exclusion on shared data.
- Make all synchronization explicit to the programmer.
- Allow libraries to operate on shared and local data.
- Represent data and access memory exactly as single-threaded programs do.
- Allow accessing local data without synchronization.
- Avoid interprocedural analysis.

5.1.1 Multithreading Terms

To support multithreading, we add three primitives and one statement form to Cyclone. The primitives have Cyclone types, so we can implement them entirely with a library written in C.

The `spawn` function takes a function pointer, a pointer to a value, and the size of the value. Executing `spawn(e1,e2,e3)` evaluates `e1`, `e2`, and `e3` to some `f`, `p`, and `sz` respectively; copies `*p` into fresh memory pointed to by some new `p2` (doing the copy requires `sz`); and executes `f(p2)` in a new thread. The spawned thread terminates when `f` returns; the spawning thread continues execution. Note that everything `*p2` points to is shared (the copy is shallow), but `*p2` is local to the new thread.

The `newlock` function takes no arguments and returns a fresh lock. Locks mediate access to shared data: for each shared object, there is a lock that a thread must hold when accessing the object. As explained below, the type system makes the connection between objects and locks.

The `nonlock` constant serves as a pseudolock. Acquiring `nonlock` has no run-time effect. Its purpose is to provide a value when a real lock is unnecessary because the corresponding data is local.

```
void inc(int* p) {
  *p = *p + 1;
}
void inc2(lock_t plk, int* p) {
  sync(plk) inc(p);
}
struct LkInt { lock_t plk; int* p; };
void g(struct LkInt* s) {
  inc2(s->plk, s->p);
}
void f() {
  lock_t lk = newlock();
  int*   p1 = new 0;
  int*   p2 = new 0;
  struct LkInt* s = new LkInt{.plk=lk, .p=p1};
  spawn(g, s, sizeof(struct LkInt));
  inc2(lk, p1);
  inc2(nonlock, p2);
}
```

Figure 5.1: Example: Multithreading Terms with C-Like Type Information

Finally, the statement `sync(`e`)`s evaluates e to a lock (or `nonlock`), acquires the lock, executes s, and releases the lock. Only one thread can hold a lock at a time, so the acquisition may *block* until another thread releases the lock. Note that nothing in Cyclone prevents deadlock.

Figure 5.1 uses these constructs but includes only the type information we might expect in C; it is not legal Cyclone. Because `inc` accesses `*p`, callers of `inc` should hold the appropriate lock if `*p` is shared. No lock is needed to call `inc2` so long as `plk` is the lock for `*p`. The function `f` spawns a thread with function `g`, lock `lk`, and pointer `p1`. Both threads increment `*p1`, but `lk` mediates access. Finally, `p2` is thread-local, so it is safe to pass it to `inc2` with `nonlock`. (We could also just call `inc(p2)`.)

5.1.2 Multithreading Types

The key extension to the Cyclone type system is *lock names*, which are, with one exception, type-level variables that describe run-time locks. Lock names do not exist at run-time. A lock has type `lock_t<`ℓ`>` where ℓ is a lock name. The key

```
void inc<ℓ::LU>(int*ℓ p ;{ℓ}) {
  *p = *p + 1;
}
void inc2<ℓ::LU>(lock_t<ℓ> plk, int*ℓ p ;{}) {
  sync(plk) inc(p);
}
struct LkInt {<ℓ::LS> lock_t<ℓ> plk; int*ℓ p; };
void g<ℓ::LU>(struct LkInt*ℓ s ;{ℓ}) {
  let LkInt{<ℓ'> .plk=lk, .p=ptr} = *s;
  inc2(lk, ptr);
}
void f(;{}) {
  let     lk<ℓ> = newlock();
  int*ℓ   p1    = new 0;
  int*loc  p2   = new 0;
  struct LkInt*loc s = new LkInt{.plk=lk, .p=p1};
  spawn(g, s, sizeof(struct LkInt));
  inc2(lk, p1);
  inc2(nonlock, p2);
}
```

Figure 5.2: Example: Correct Multithreaded Cyclone Program

restriction is to include lock names in pointer types, for example `int*`ℓ. We allow dereferencing a pointer of this type only where the type-checker can ensure that the thread holds a lock with type `lock_t<`ℓ`>`. The absence of data races relies on only one such lock existing.

Thread-local data fits in this system by having a special lock name *loc*. We give `nonlock` the type `lock_t<`*loc*`>` and annotate pointers to thread-local data with *loc*. We always allow dereferencing such pointers; we never let them be reachable from an argument to `spawn`.

Like type variables, lock names other than *loc* must be in scope. We can introduce lock names via *universal quantification*, *existential quantification*, or *type constructors*, all of which capture important idioms.

Functions universally quantify over lock names so callers can pass pointers with different lock names. For example, Figure 5.2 has all the type information omitted from Figure 5.1, including several annotations that are unnecessary due to defaults. We can instantiate the `inc` and `inc2` functions using any lock name for ℓ. (Section 5.1.3 explains the kind annotations `LS` and `LU`.) Instantiation is

implicit. As examples, the first use of `inc2` in `f` instantiates ℓ with the ℓ in the type of `p1` whereas the second instantiates ℓ with *loc*.

Each function type has an *effect*, a set of lock names (written after the parameters) that callers must hold. In our example, each function has the empty effect (`{}`, which really means {*loc*}), except `inc` and `g`. Effects are the key to enforcing the locking discipline: Each program point is assigned an effect—the current capability. A function-entry point has the function's effect. Every other statement inherits the effect of its enclosing statement except for `sync (`e`)` s: If e has type `lock_t<`ℓ`>`, then `sync (`e`)` s adds ℓ to the current capability for s. If e has type $\tau*\ell$, then we allow $*e$ only where ℓ is in the current capability. Similarly, a function call type-checks only if the current capability (after instantiation) is a superset of the callee's effect. For example, the call to `inc` in `inc2` type-checks because the caller holds the necessary lock.

The type of `newlock()` is $\exists\ell$:`LS.lock_t<`ℓ`>`; there exists a lock name such that the lock has that name. As usual, we *unpack* a value of existential type before using it. The declaration `let lk<`ℓ`> = newlock();` in `f` is an unpack. It introduces variable `lk` and lock name ℓ. Their scope is the rest of the code block. `lk` is bound to the new lock and has type `lock_t<`ℓ`>`. We could unpack a lock multiple times (e.g., with names ℓ_1 and ℓ_2), but acquiring the lock via a term with type `lock_t<`ℓ_1`>` would not permit dereferencing pointers with lock name ℓ_2.

Existentials are important for user-defined types too. The type `struct LkInt` is an example: Pointer `p` has the same lock name as lock `plk`. This name is existentially bound in the type definition. As with `newlock()`, using a `struct LkInt` value requires an unpack, as in `g`. This pattern form binds `lk` to `s->plk` (giving `lk` type `lock_t<`ℓ'`>`) and `ptr` to `s->p` (giving `ptr` type `int*`ℓ'). To form a `struct LkInt` value, such as in `f`, the fields' types must be consistent with respect to their (implicit) instantiation of ℓ.

As noted earlier, existential types are a good example of the need for mutual exclusion. Suppose two threads share a location of type `struct LkInt`. As in C, one thread could mutate the `struct` by assigning a different `struct LkInt` value, which could hold a different lock. This mutation is safe only if no thread uses the shared `struct` during the mutation (at which point perhaps `plk` has changed but `p` has not).

Finally, type definitions can have lock-name parameters. For example, for a list of `int*` values, we could use:

```
struct Lst<ℓ1::LU,ℓ2::LU> {
  int*ℓ1 hd;
  struct Lst<ℓ1,ℓ2> *ℓ2 tl;
};
```

This defines a type constructor that, when applied to two lock names, produces a type. For thread-local data, `struct Lst<`*loc*`,`*loc*`>` is a good choice. With universal quantification, functions for lists can operate over thread-local or thread-shared data. They can also use different locking idioms. Here are some example prototypes:

```
int length<ℓ1::LU,ℓ2::LU>(struct Lst<ℓ1,ℓ2> ;{ℓ2});
int sum<ℓ1::LU,ℓ2::LU>(struct Lst<ℓ1,ℓ2> ;{ℓ1,ℓ2});
int sum2<ℓ1::LU,ℓ2::LU>(struct Lst<ℓ1,ℓ2>, lock_t<ℓ2> ;{ℓ1});
void append<ℓ1::LU,ℓ2::LU,ℓ3::LU>(struct Lst<ℓ1,ℓ2>, struct Lst<ℓ1,ℓ3>
                                   ;{ℓ2,ℓ3});
```

For `length` (which we suppose computes a list's length), the caller acquires the lock for the list spine and `length` does not access the list's elements. We also use a caller-locks idiom for `sum`, whereas `sum2` uses a hybrid idiom in which the caller acquires the elements' lock and `sum2` (presumably) acquires the spine's lock. Finally, we suppose `append` mutates its first argument by appending a copy of the second argument's spine. The two lists can have different lock names for their spines precisely because `append` copies the second spine. Like `length`, the elements are not accessed.

5.1.3 Multithreading Kinds

We have used several now-familiar typing technologies to ameliorate the restrictions that lock names impose. These techniques apply naturally because we treat lock names as types that describe locks instead of values. We use *kinds* to distinguish "ordinary" types from lock names. In this sense, lock names have kind `L` and other types have kind `A`.

Kinds also have *sharabilities*, either `S` (for sharable) or `U` (for possibly unsharable). The lock name for the lock `newlock` creates has kind `LS` whereas *loc* has kind `LU`. Kind `LS` is a *subkind* of `LU`, so every lock name has kind `LU`. We use subsumption to check the calls `inc2(lk, ptr)` and `inc2(lk, p1)` in our example.

We use sharabilities to prevent thread-local data from being reachable from an argument passed to `spawn`: Memory kinds also have sharabilities. For example, τ`*`ℓ has kind `AS` only if τ has kind `AS` and ℓ has kind `LS`. In general, a type of kind `AS` cannot contain anything of kind `LU`. As expected, `AS` is a subkind of `AU`.

With a bit of polymorphism, we can give `spawn` the type:

```
void spawn<α::AS,ℓ::LU>(void f(α*loc; {}), α*ℓ, sizeof_t<α>; {ℓ});
```

Kinding ensures all shared data uses locking. The effect of `f` is `{}` because new threads hold no locks. The effect of `spawn` is {ℓ} because it copies what the second

argument points to. As Section 3.2 explains, the only value of type `sizeof_t<`α`>` is `sizeof(`α`)`, so the type system ensures callers pass the correct size.

In our example, we instantiate the α in `spawn`'s type with `struct LkInt`, which has type `AS` only because the existentially bound lock name in its definition has kind `LS`. A term like `LkInt{.plk=nonlock, .p=p2}` is ill-formed because `nonlock` has type `lock_t<`*loc*`>`, but `struct LkInt` requires a lock name of kind `LS`.

5.1.4 Default Annotations

The type system so far requires a lock name for every pointer type and lock type, and an effect for every function. We can extend our simple techniques for omitted type information to make the vast majority of these annotations optional.

First, when a function's effect is omitted, it implicitly includes all lock names appearing in the parameters' types. Hence the default idiom is "caller locks." Second, lock names are always optional. How they are filled in depends on context:

- Within function bodies, a unification engine can infer lock names.
- Within type definitions, we use *loc*.
- For function parameter and return types, we can generate fresh lock names (and include them in default effects). We discuss below several options for how many lock names to generate. Top-level functions implicitly universally quantify over free lock names.

Third, the default sharability for kinds is `U`.

All inference remains intraprocedural. The other techniques fill in defaults without reference to function bodies. Hence we can maintain separate compilation.

Different strategies for generating omitted lock names in function prototypes have different benefits. First, we could generate a different lock name for each unannotated pointer type. This strategy makes the most function calls type-check. However, if the prototype has no explicit locking annotations, the function body will not type-check if it returns a parameter, assigns one parameter to another, has a local variable that might hold different parameters, etc. We had similar problems in Chapter 4, but we could use region subtyping to give region annotations to local variables. Second, we could exploit region annotations in the prototype by using the same lock name for pointer types with the same (explicit) region name. This refinement of the first strategy takes care of function bodies that do things like return parameters. It does not always suffice for bodies that use region subtyping because lock names do not enjoy subtyping. Furthermore, callers cannot pass objects that are in the same region but guarded by different locks. Third, we

could just use *loc* for omitted lock names. This solution has the advantage that single-threaded programs type-check as multithreaded programs, unless they use global variables. (As Section 5.5 discusses, global variables require locks.) However, it means programmers must use extra annotations to write code that is safe for multithreading, even when callers acquire locks.

Because these strategies are all useful, Cyclone should support convenient syntax for them. One possibility is a pragma that changes the strategy, but pragmas that change the meaning of prototypes can make programs more difficult for humans to understand. We could make a similar argument for a pragma to make the default region annotation ρ_H in prototypes.

In our example, the first or second strategy and the other techniques allow the following abbreviated prototypes:

```
void inc(int* p);
void inc2(lock_t<ℓ> plk, int*ℓ p; {});
struct LkInt {<ℓ:LS> lock_t<ℓ> plk; int*ℓ p; };
void g(struct LkInt* s);
void f();
```

The lock names for variables `p1`, `p2`, and `s` are also optional.

5.2 Interaction With Type Variables

We must resolve two issues to use type variables in multithreaded Cyclone:

1. How do we prevent thread-local data (data guarded by *loc*) from becoming thread-shared?

2. How do we extend effects to ensure that polymorphic code uses proper synchronization?

We sketched our solution to the first issue in the previous section: A type's kind includes a sharability (`S` or `U`) in addition to `B` versus `A`. Sharability `S` means a type cannot describe thread-local data. The actual definition is inductive over the type syntax: Sharability `S` means no part of the type has kind `BU`, `AU`, or `LU`. Combining the two parts of a type's kind, we have richer subkinding on types: `BS` $\leq$ `BU`, `AS` $\leq$ `AU`, `BS` $\leq$ `AS`, `BU` $\leq$ `AU`, `BS` $\leq$ `AU`, and `LS` $\leq$ `LU`. Sharability `S` is necessary only for using `spawn`, so almost all code uses sharability `U`.

To extend effects, consider the function `app` that calls parameter `f` with parameter `x` of type α: Its effect should be the same as the effect for `f`, but how can we describe the effect for `f` when all we know is that it takes some α? If we give

`f` and `app` the effect `{}`, then `app` is unusable for thread-shared data: `f` cannot assume it holds any locks, and the caller passes it none to acquire.

Our solution introduces `locks(`τ`)`, a new form of effect that represents the effect consisting of all lock names and type variables occurring in τ. We can write a polymorphic `app` function like this:

```
void app<α::BU>(void f(α; locks(α)), α x; locks(α)) { f(x); }
```

If we instantiate α with `int*`ℓ_1`*`ℓ_2, then the effect means we can call `app` only if we hold `locks(int*`ℓ_1`*`ℓ_2`)={`ℓ_1`,`ℓ_2`}`. As another example, if a polymorphic function calls `app` using β`*`ℓ for α, the current capability must include `locks(`β`)` and ℓ.

Including `locks(`α`)` in the effect of a function type that universally quantifies over α describes a "caller locks" idiom. As described in Section 5.1.4, this idiom is what we want if programmers omit explicit effects. Hence the default effect for a polymorphic function includes `locks(`α`)` for all type parameters α. In our `app` example, we can omit both effects. In fact, by making `B` and `A` short-hand for `BU` and `AU`, polymorphism poses no problem for type-checking single-threaded code as multithreaded code.

However, we cannot yet write polymorphic code using a "callee locks" idiom, such as in this wrong example:

```
void app2<α::BU,ℓ::LU>(void f(α; locks(α)), α x, lock_t<ℓ> lk; {}){
  sync lk { f(x); }
}
```

We want to call `app2` with no locks held because it acquires `lk` before calling `f`. But nothing expresses any connection between `{`ℓ`}` (the capability where `app2` calls `f`) and `locks(`α`)` (the effect of `f`).

Our solution enriches function preconditions with *constraints* of the form $\epsilon_1 \subseteq \epsilon_2$ where ϵ_1 and ϵ_2 are effects. As in Chapter 4, the constraint means, "if ϵ_2 is in the current capability, then it is sound to include ϵ_1 in the current capability."

For example, we can write:

```
void app2<α::BU,ℓ::LU>(void f(α; locks(α)), α x, lock_t<ℓ> lk; {}
                       : locks(α)⊆{ℓ}) {
  sync lk { f(x); }
}
```

At the call to `f`, we use the current capability (`{`ℓ`}`) and the constraint to cover the effect of `f` (`locks(`α`)`, which we can omit). Callers of `app2` must establish the constraint by instantiating α and ℓ with some τ and ℓ' respectively such that we know `locks(`τ`)={}` or `locks(`τ`)={`ℓ'`}`. To support instantiating α with some τ that needs more (caller-held) locks, we can use this more sophisticated type:

```
void app2<α::BU,ℓ::LU,β::AU>(void f(α), α x, lock_t<ℓ> lk; locks(β)
                             : locks(α) ⊆ {ℓ}∪locks(β));
```

In summary, polymorphism compelled us to add a way to describe the lock names of an unknown type (`locks(`α`)`) and a way to constrain such lock names (`locks(`α`)`$\subseteq \epsilon$). With these features, we can express what locks a thread should hold to use a value of unknown type. By our choice of default effect, programmers can usually ignore these additions. They are needed for polymorphic code using "callee locks" idioms. Dually (though we did not show an example), we need them to use existential types with "caller locks" idioms.

5.3 Interaction With Regions

So far, we have described multithreaded Cyclone as if data were never deallocated. Garbage collection can maintain this illusion, but the region system presented in Chapter 4 gives programmers finer control. In this section, we describe how the region system is analogous to the locking system and how combining the systems allows threads to share reclaimable data.

5.3.1 Comparing Locks and Regions

The correspondence between the static type systems for regions and locking is striking and fascinating. We use singleton types for locks and handles, type variables (of different kinds) for decorating pointer types, `locks(`α`)` and `regions(`α`)` for describing requirements for abstract types, `sync` and `region` for gaining access rights, *loc* and ρ_H for always available resources, constraints for revealing partial information about abstract types, and so on.

There are compelling reasons for the depth of the analogy. Accessing memory safely requires that the appropriate region is live and the appropriate lock is held. Type variables, pointer-type annotations, and effects capture both aspects of access rights in the same way: It is safe to dereference a pointer of type $\tau * \rho\ell$ if the current capability includes ρ and ℓ. At this level, the type system is oblivious to the fact that ρ names a region and ℓ names a lock; the notion of access rights is more abstract. For both systems, the constructs for amplifying rights (`region` and `sync`) increase the current capability for a lexically scoped statement. Lexical scope simplifies the rules for determining the current capability, but it is not essential.

Most differences between the region and locking systems are by-products of a natural distinction: A region can be allocated and deallocated at most once, but a lock can be acquired and released multiple times. Therefore, there is little reason to separate region creation from the right to access the region. On the other hand, the

locking system separates `newlock` from `sync`. The region lifetime orderings induce a natural notion of subtyping, so the `region` construct introduces a compile-time constraint. Because we can acquire locks multiple times, the locking system has no such subtyping. Put another way, regions have a fixed ordering that locks do not, so we allow programs of the form `sync lk1 {sync lk2 {`s_1`;}}; sync lk2 {sync lk1 {`s_2`;}}`. (However, well-known techniques for preventing deadlock impose a partial order on locks [73, 31].)

The more complicated kind system for locks arises from the difference between *loc* and ρ_H. For both, it is always safe to access memory guarded by them. However, there are no restrictions on using ρ_H whereas *loc* must actually describe thread-local data. If we restricted ρ_H, for example to prevent some space leaks when not using a garbage collector, the kind system for regions might become more sophisticated.

5.3.2 Combining Locks and Regions

The basic constructs for regions and locking compose well: Pointer types carry region names *and* lock names. Accessing memory requires its region is live *and* its lock is held. Continuing an earlier example, `app` could have this type:

`void app(void f(`α`; regions(`α`),locks(`α`))), `α`; regions(`α`),locks(`α`));`

Moreover, by combining the rules for default annotations, it suffices to write:

`void app(void f(`α`), `α`);`

The only interesting interaction is ensuring that one thread does not access a region after another thread deallocates it.

First, we impose a stricter type for `spawn`. To prevent the spawned thread from accessing memory the spawning thread deallocates, we use a *region bound* to ensure that the shared data can reach only the heap: For `spawn` (which we recall uses α to quantify over the type its second argument points to), we add the region-bound precondition `regions(`α`)`$< \rho_H$. This solution is sound, but it relegates all thread-shared data to the heap.

To add expressiveness, we introduce the construct `rspawn`. We type-check `rspawn(e1,e2,e3,e4)` like `spawn(e1,e2,e3)` except we quantify over a region name ρ, change the precondition to `regions(`α`)`$< \rho$, and require `e4` to have type `region_t<`ρ`>`. In other words, the new argument is a handle indicating the shared value's region bound. There is still no way to share a stack pointer between threads. Doing so safely would impose overhead on using local variables, which C and Cyclone programmers expect to be very fast.

If a handle is used in a call to `rspawn`, then the corresponding region will live until the spawning thread would have deallocated it *and* the spawned thread terminates. The next section explains how the run-time system maintains this invariant. The remaining complication is subtyping: As Section 4.1.4 explains, Cyclone allows casting $\tau * \rho_1$ to $\tau * \rho_2$ so long as ρ_1 outlives ρ_2. But that means we also cannot deallocate the region named ρ_1 until all threads spawned with the handle for ρ_2 have terminated. If ρ_1 is a dynamic region, the run-time system can support this added complication efficiently, but ρ_1 should not be a stack region.

To prevent casting stack pointers to dynamic-region pointers used in calls to `rspawn`, we enrich region kinds with sharabilities `S` and `U` (as with other kinds), as well as a sharability `D` for "definitely not sharable." Both `RS` and `RD` are subkinds of `RU`. A stack-region name always has kind `RD`. The programmer chooses `RS` or `RU` for a dynamic-region name. If region name ρ_1 describes a live region at the point the region named ρ_2 is created, we introduce $\rho_1 < \rho_2$ only if ρ_2 has kind `RD` or ρ_1 has kind `RS`. The handle passed to `rspawn` must have a region name of kind `RS`. Single-threaded programs can choose `RD` for all dynamic-region names.

5.4 Run-Time Support

The run-time support for Cyclone's basic thread operations is simple. If garbage collection is used for the heap region, then the collector must, of course, support multithreading. The `newlock`, `sync`, and `spawn` operations are easy to translate into operations common in thread packages such as POSIX Threads [35]. We translate `nonlock` to a distinguished value that `sync` checks for before trying to acquire a lock. The cost of this check is small, less than the check required for reentrant locks. (We could add a kind `LD` that does not describe *loc* and use this kind to omit checks for `nonlock`, but the complication seems unnecessary.)

Non-local control (jumps, return statements, and exceptions) are a minor complication because a thread should release the lock that a `sync` acquired when control transfers outside the scope of the `sync`. For jumps and return statements, the compiler can insert the correct lock releases (with checks for `nonlock`). For exceptions, we must maintain a per-thread run-time list of locks acquired after installing an exception handler.

The interesting run-time support is the implementation of `rspawn` because we must not deallocate a region until every thread is done with it. To have the necessary information, every dynamic-region handle contains a list of live threads using it (including the thread that created it). Also, each thread has a list of the live dynamic-region handles it has created. The list is sorted by lifetime. These lists are (internally) thread-shared, so the run-time system uses locks for mediating

access to them. We maintain the lists as follows:

1. `rspawn`: Before starting the spawned thread, add it to the handle's thread list. After the spawned thread terminates, remove it from the handle's thread list. If the handle's thread list is now empty and the handle is last (youngest) in its handle list, deallocate the region, remove the handle from its handle list, and recur on the next (older) handle in the handle list.

2. `region r` s: Before executing s, create a region, add its handle to the (young) end of the thread's handle list, and add the executing thread to the handle's thread list. When control leaves s, remove the executing thread from the handle's thread list. If the handle's thread list is now empty and the handle is last (youngest) in its handle list, deallocate the region and remove the handle from its handle list.

The dynamic regions that a thread creates continue to have last-in-first-out lifetimes. However, stack regions might be deallocated before some dynamic regions created after them, which is why sharabilities restrict region subtyping. Note that if the lists are doubly linked, we add only $O(1)$ amortized cost to `rspawn` and `region`.

5.5 Evaluation

This section informally evaluates the strengths and weaknesses of Cyclone's support for multithreading. Most of the strengths have already been mentioned, but it is still useful to summarize them together. Some of the weaknesses are analogous to region-system weaknesses, but others amount to a lack of support for sound synchronization idioms other than lock-based mutual exclusion.

5.5.1 Good News

Because data races could compromise Cyclone's safety, the type system prevents them. Because race-prevention is mostly static, it does not hurt run-time performance. Most importantly, multithreaded programs read and write memory locations just like single-threaded programs. An alternative safe design would be to generate code for memory access that acquired the lock, performed the access, and released the lock. Performance could suffer, and any optimizations to reduce the number of lock operations would be beyond programmers' control. One way to describe Cyclone's `sync` operation and effect system is to say that programmers do their own optimizations by hoisting lock acquisitions and assigning locks

to memory locations. The type system prevents errors, but disallows some safe optimizations.

Explicit function effects keep analysis intraprocedural while allowing caller-locks, callee-locks, and hybrid idioms. Because caller-locks idioms produce the simplest, most efficient single-threaded code (there are fewer lock acquisitions and less lock passing), the default effects encode this idiom. However, this decision means functions that acquire locks they are passed invariably need explicit effects. They would type-check with the default effect, but then they could only be called in contexts where they are likely to deadlock. (If locks are reentrant, they would not deadlock, but acquiring locks would then be useless.)

The notion of thread-local data supports the "special case" where a memory location is never reachable to any thread except the one that creates it. Because race conditions on such memory are impossible, no lock is necessary. In many multithreaded applications, most memory remains thread-local, so Cyclone aims to impose as little burden as possible for this case. One solution would be to make the default lock name always be *loc*. For function parameters, this solution is less burdensome than fresh lock names that are in the implicit effect, so we would only be restricting functions' usefulness. Within function bodies, intraprocedural inference can require fewer annotations than assuming *loc*. Its design should ensure that it never requires more annotations. Ultimately, an objective evaluation is that single-threaded programs type-check as multithreaded code.

The kind system remains simple enough that there are only a few kinds, but powerful enough that we can give `spawn` a Cyclone type. Moreover, subkinding lets programmers write code once and use it for thread-local and thread-shared data. Because thread-local data is the common case, by default we assume function parameters might be thread-local. Therefore, kind annotations are necessary for terms that might become reachable from arguments to `spawn`. The `nonlock` term is a simple trick for allowing clients to use callee-locks code with thread-local data.

Finally, we have already explained in detail how the thread system interacts smoothly with parametric polymorphism and region-based memory management. A small disadvantage is that sharable regions may outlive the construct that creates them. Nonetheless, programmers desiring stronger memory-reclamation assurances can declare dynamic regions to be unsharable. Another disadvantage is that the run-time system must maintain more region information and use synchronization on it. However, we should expect the run-time system for a multithreaded language to incur some synchronization overhead.

5.5.2 Bad News

As a sound, decidable type system, Cyclone's data-race prevention is necessarily conservative, forbidding some race-free programs. Here we describe a few of the more egregious limitations and how we might address them.

Thread-shared data that is *immutable* (never mutated) does not need locking. Expressing this read-only invariant is straightforward if we "take `const` seriously" (unlike C), but qualifier polymorphism [81] becomes important for code reuse. Similarly, *reader/writer* locks allow mutation and concurrent read access. Annotating pointer types with read and write locks should pose no technical problems. In short, the type system assumes any read of thread-shared data requires exclusive access, but immutability and reader/writer locks are safe alternatives.

Global variables are thread-shared, so they require lock-name annotations. But that means we need locks and lock names with global scope. Worse, single-threaded programs with global variables do not type-check as multithreaded programs because they need lock names. Note that thread-local variables with thread-wide scope are no problem.

Oftentimes, thread-shared data has an *initialization phase* before it becomes thread-shared. During this phase, locking is unnecessary. A simple flow analysis will probably suffice to allow access without locking so long as an object could not yet have become shared. We can support a trivial but very common case: When allocating and initializing data (e.g., with `new`) guarded by ℓ, it is not necessary to hold ℓ. Incorporating a flow analysis obtains the flexibility that Chapter 6 provides for initialization.

Data objects sometimes *migrate* among threads without needing locking. An example is a producer/consumer pattern: a producer thread puts objects in a shared queue and a consumer thread removes them. If the producer does not use objects after enqueuing them, the objects do not need locks. This idiom is safe because of restricted aliasing (the producer does not use other retained references to the object), so the type system presented here is ill-equipped to support it. The analogy with memory management continues here: It is safe to call `free` precisely when subsequent computation does not use other retained references. Therefore, any technology suitable for supporting safe uses of `free` should be suitable for supporting object migration. Indeed, related work that permits object migration generally distinguish "unique pointers," which the type system ensures are the only pointers to the object to which they point.

Synchronization mechanisms other than mutual-exclusion locks often prove useful. Examples include semaphores and signals. It is also important to expose some implementation details, such as whether a lock is a spin-lock or not. In general, Cyclone should support the mechanisms that a well-designed threads library for

C (such as POSIX Threads [35]) provides. Libraries do not require changing the language, but the compiler cannot enforce that clients use such libraries safely.

The thread system has many of the same limitations as the region system, but the limitations may be less onerous in practice. For example, locks are held during execution that corresponds to lexical scope. Therefore, there is no way for a callee to release a lock that a caller acquires (which could reduce parallelism because other threads are blocked) or for a callee to acquire locks that a caller releases (which could allow more flexible library interfaces). Java has the same restriction; I have not encountered substantial criticism of this design decision.

The type system also suffers the same lack of principal typing as Chapter 4 describes. Possible solutions are analogous. For example, pointer types could carry effects. Dereferencing pointers would require the current capability implied the entire effect.

Some other shortcomings deserve brief mention. First, the annotation burden for reusable type constructors increases with threads. To ameliorate the problem, we could support type-level variables that stood for a region name *and* a lock name. That is, we could write τ`*`α where α abstracted a region and a lock, rather than τ`*`$\rho\ell$. A similar combination might prove useful at the term level: We could have regions for which all objects in the region had the same lock and allow the region handle to serve as the lock also. Separating regions and locks is more powerful, but merging them is often convenient.

Second, the type system does not support guarding different fields of the same `struct` with different locks. Here, the analogy with regions breaks down because it makes no sense for different fields of the same `struct` to have different lifetimes. The main problem with supporting different locks for different fields is how to annotate pointer types.

Third, abstract types (e.g., `struct Foo;`) need explicit sharability annotations unless we assume they are all unsharable. The problem is more pronounced for abstract type constructors: For `struct Bar<`α`::`κ`>`, an explicit annotation should mean an application of the type constructor is sharable if all the arguments to it are sharable. Essentially, we need to leak whether the implementation has any unsharable fields (i.e., any uses of *loc*). In Chapter 4, we did not have this problem because hidden uses of ρ_H do not restrict how a client can use an abstract type.

Finally, it bears repeating that we do not prevent deadlock (although the type system is compatible with reentrant locks, which help a bit). Deadlock is undesirable, but it does not violate type safety.

5.6 Formalism

This section defines a formal abstract machine and a type system for it that capture most of Cyclone's support for multithreading. The formalism is very much like the formalisms in earlier chapters, which supports the claim that similar techniques prevent different safety violations. As such, we focus on how this chapter's abstract machine differs from earlier ones, beginning with a summary of the most essential differences.

First, to compensate for the complications that threads introduce, we make some simplifications. We do not integrate memory management, so all objects live forever, as in Chapter 3. We forbid quantification over types of unknown size, as in Chapter 4. We do not allow $e.i$ as a left-expression, so it is not possible to take the address of a field or assign to part of an aggregate object. However, if x holds a pair, it is easy to simulate assigning to a field via $x{=}(v, x.1)$ or $x{=}(x.0, v)$.

Second, a machine state includes multiple threads. Thread scheduling is nondeterministic: Any thread capable of taking a step might do so. Each thread comprises a statement (for its control) and a set of locks that it currently holds. The machine also has a set of available locks (held by no thread) and a single shared heap. The type-safety proof uses a type system that partitions the heap into thread-local portions for each thread and a thread-shared portion that is further divided to distinguish portions guarded by locks held by different threads. This partitioning is purely a proof technique. The abstract machine has one "flat" heap and there is no run-time information ascribing locks to locations. To contrast, in Chapter 4, regions existed at run-time.

Third, the assignment $x{=}v$ takes two steps, and x holds the expression junk_v after the first step. If a thread reads this junk value, it might later become stuck because the dynamic semantics does not allow destructing junk_v. Because the type system prevents data races, reading junk is impossible.

Fourth, the kind system includes sharabilities for reasons explained earlier in this chapter. Because the formalism does not include regions, we do not include a "definitely not sharable" sharability.

Finally, despite striking similarities between the constructs for regions in Chapter 4 and locks in this chapter, the creation of locks and the scope of lock names is different. In Chapter 4, statements that created locations or regions included a binding occurrence of a region name that was in scope for a subsequent statement. In this chapter, statements that create locations include a bound occurrence of a lock name (that is already in scope, of course). Similarly, a sync statement acquires a lock that already exists. Given the discussion in Section 5.3.1, these differences are exactly what we should expect.

$$
\begin{array}{rrcl}
\text{kinds} & \sigma & ::= & \mathtt{S} \mid \mathtt{U} \\
 & \theta & ::= & \mathtt{B} \mid \mathtt{A} \mid \mathtt{L} \\
 & \kappa & ::= & \theta\sigma \\
\text{effects} & \epsilon & ::= & \emptyset \mid \alpha \mid i \mid \epsilon \cup \epsilon \\
\text{constraints} & \gamma & ::= & \cdot \mid \gamma, \epsilon \subseteq \epsilon \\
\text{types} & \tau, \ell & ::= & \alpha \mid \text{int} \mid \tau \times \tau \mid \tau \xrightarrow{\epsilon} \tau \mid \tau * \ell \mid \forall \alpha{:}\kappa[\gamma].\tau \mid \exists \alpha{:}\kappa[\gamma].\tau \\
 & & \mid & \text{lock}(\ell) \mid \text{S}(i) \mid \mathit{loc} \\
\text{terms} & s & ::= & e \mid \mathsf{return}\ e \mid s; s \mid \mathsf{if}\ e\ s\ s \mid \mathsf{while}\ e\ s \mid \mathsf{let}\ \ell, x{=}e;\ s \\
 & & \mid & \mathsf{open}\ e\ \mathsf{as}\ \ell, \alpha, x;\ s \mid \mathsf{sync}\ e\ s \mid s; \mathsf{release}\ i \mid \mathsf{spawn}\ e(e) \\
 & e & ::= & x \mid i \mid f \mid \&e \mid *e \mid (e, e) \mid e.i \mid e{=}e \mid e(e) \mid \mathsf{call}\ s \\
 & & \mid & e[\tau] \mid \mathsf{pack}\ \tau, e\ \mathsf{as}\ \tau \mid \mathsf{nonlock} \mid \mathsf{newlock}() \mid \mathsf{lock}\ i \mid \mathsf{junk}_v \\
 & f & ::= & (\tau, \ell\ x) \xrightarrow{\epsilon} \tau\ s \mid \Lambda\alpha{:}\kappa[\gamma].f \\
\text{values} & v & ::= & i \mid \&x \mid f \mid (v, v) \mid \mathsf{pack}\ \tau, v\ \mathsf{as}\ \tau \mid \mathsf{nonlock} \mid \mathsf{lock}\ i \\
\text{heaps} & H & ::= & \cdot \mid H, x \mapsto v \mid H, x \mapsto \mathsf{junk}_v \\
\text{locks} & L & ::= & \cdot \mid L, i \\
 & \overline{L} & ::= & L; L; L \\
\text{threads} & T & ::= & L; s \\
\text{states} & P & ::= & L; L; H; T_1 \cdots T_n \\
\text{contexts} & \Delta & ::= & \cdot \mid \Delta, \alpha{:}\kappa \\
 & \Gamma & ::= & \cdot \mid \Gamma, x{:}(\tau, \ell) \\
 & C & ::= & L; \Delta; \Gamma; \gamma; \epsilon
\end{array}
$$

Figure 5.3: Chapter 5 Formal Syntax

5.6.1 Syntax

Figure 5.3 presents the language's syntax. We focus on the constructs most relevant to multithreading.

Kinds include a θ for distinguishing types of known size (B), types of unknown size (A), and lock names (L). Kinds also include sharabilities: sharability S indicates that no part of the type describes thread-local data. In source programs, only type variables and *loc* can have kinds of the form $\mathtt{L}\sigma$. In particular, *loc* has kind LU. At run-time, we name actual locks with integers. To describe the lock i, we use the lock name $\text{S}(i)$, which has kind LS. The term $\mathsf{lock}\ i$ is how programs refer to the lock i. The type of $\mathsf{lock}\ i$ is $\text{lock}(\text{S}(i))$, which has kind AS. If we know a type has kind LS or LU, we often write ℓ instead of τ to remind us.

Effects and constraints are exactly like in Chapter 4, except they represent lock sets and inequalities among them instead of regions sets and outlives relationships.

In particular, the only way $i \subseteq i'$ can hold is if $i = i'$. As such, constraints are useful only with type variables (e.g., $\alpha \subseteq \epsilon$). As in Chapter 4, we implicitly identify effects up to set equality, including associativity, commutativity, and idempotence.

As expected, quantified types can introduce constraints, function types include explicit effects, and types for pointers and locks include lock names.

Most statement forms are just like in earlier chapters. The let and open forms specify a lock name ℓ that guards the location x these statements allocate. The lock name must already be in scope. Access to x requires the current capability and constraints imply the executing thread holds ℓ. The term sync e s evaluates e to a lock, acquires the lock (potentially remaining stuck until another thread releases the lock), executes s, and releases the lock. To remember which lock to release, sync (lock i) s evaluates to s; release i (provided i is available).

The last statement form, spawn $e_1(e_2)$ evaluates e_1 and e_2 to a function and a value and creates a new thread to evaluate the function called with the value. Unlike actual Cyclone, we do not require that the size of the passed value is known. This version of spawn is not implementable as a C function, but it is simpler.

The novel expression forms include nonlock (a pseudolock for thread-local data), newlock() (for creating a fresh lock), and lock i (a form inappropriate for source programs that describes a lock). The form junk_v is also inappropriate for source programs. The machine uses it when mutating a heap location to v. We include v so the machine knows what value should be written when the thread performing the mutation takes another step.

A lock set L is implicitly reorderable. (Unlike the region sets R in Chapter 4, we do not use lock sets to encode orderings because locks have no outlives-like relationship.) When a thread takes a step, it might use or modify three lock sets: the set of all locks the program has created, the set of locks held by no thread, and the set of locks held by the thread itself. When the form of these three sets is unimportant, we abbreviate them with $\overline{L}$. The thread $L; s$ holds exactly the locks in L and executes s. A program state $L; L_0; H; T_1 \cdots T_n$ includes the set of all created locks (L), the set of available locks (L_0), one heap (H), and n threads. The explicit L is redundant because it should always be the union of L_0 and the lock sets in each thread, but it is technically convenient to maintain it separately.

Some final technical considerations are analogous to similar ones in Chapter 4: A type context includes a set of created locks (L, always empty for source programs), the kinds of type variables (Δ), the types and lock names for term variables (Γ), the current capability (ϵ), and a collection of assumed constraints (γ). Given $C = L; \Delta; \Gamma; \gamma; \epsilon$, we write C_L, C_Δ, C_Γ, C_γ, and C_ϵ for L, Δ, Γ, γ, and ϵ, respectively. Heaps are implicitly reorderable (unlike in Chapter 4), as are contexts Δ and Γ. We use juxtaposition (e.g. HH') for the union of two maps that we assume have disjoint domains. We write $L \subseteq L'$ to mean every i in L is in L'.

$$\frac{H; (L; L_0; L_i); s_i \xrightarrow{s} H'; (L'; L'_0; L'_i); \cdot; s'_i}{L; L_0; H; T_1 \cdots (L_i; s_i) \cdots T_n \to L'; L'_0; H'; T_1 \cdots (L'_i; s'_i) \cdots T_n} \text{ DP5.1}$$

$$\frac{H; (L; L_0; L_i); s_i \xrightarrow{s} H'; (L'; L'_0; L'_i); s; s'_i}{L; L_0; H; T_1 \cdots (L_i; s_i) \cdots T_n \to L'; L'_0; H'; T_1 \cdots (L'_i, s'_i) \cdots T_n(\cdot; s)} \text{ DP5.2}$$

$$\frac{}{L; L_0; H; T_1 \cdots T_j(\cdot; \mathsf{return}\ v)T_k \cdots T_n \to L; L_0; H; T_1 \cdots T_j T_k \cdots T_n} \text{ DP5.3}$$

Figure 5.4: Chapter 5 Dynamic Semantics, Programs

5.6.2 Dynamic Semantics

The rules for rewriting P to P' (Figure 5.4) are nondeterministic because they allow any thread that can take a step (as defined by the $\xrightarrow{s}$ rules, which we describe below) to do so. Rule DP5.1 is for a thread that takes a step and does not spawn a new thread. Rule DP5.2 is for a thread that spawns a new thread when it takes a step. Rule DP5.3 is a "clean-up" rule to remove terminated threads that hold no locks. It is not necessary for type safety.

A thread can create a new lock, acquire or release a lock, change the (shared) heap, and create a new thread. Hence the single-thread evaluation rules (Figure 5.5) have the form $H; (L; L_0; L_h); s \xrightarrow{s} H'; (L'; L'_0; L'_h); s_{\text{opt}}; s'$ meaning the thread $L_h; s$ becomes $L'_h; s'$ while changing the heap from H to H', the set of created locks from L to L', and the set of available locks from L_0 to L'_0. If s_{opt} is $\cdot$, then no thread is spawned, else s_{opt} is some s'' and the new thread is $\cdot; s''$. (It starts with no threads held.)

We mention only some interesting aspects of the statement-rewriting rules. Rule DS5.1 allocates and initializes a fresh heap location. It would be more in the spirit of the abstract machine to require two steps to initialize the location, but immediate initialization is simpler because we do not need to prove that fresh locations can be accessed without synchronization. (See the discussion of initialization in Section 5.5.) Rule DS5.9 encodes the fact that acquiring nonlock has no run-time effect whereas DS5.8 applies only if the necessary lock is available. Conversely, rules DS5.10 and DS5.11 make the appropriate lock available. Rule DS5.12 is the only noninductive rule that creates a new thread. The spawned thread starts with the statement $\mathsf{return}\ v_1(v_2)$.

Figure 5.6 has the evaluation rules for right-expressions and left-expressions. The latter are simpler than in previous chapters because we do not allow left-expressions of the form $e.i$. The interesting rules are DR5.2A, DR5.2B, and DR5.8. The first two are for the two steps that mutation takes. The result of DR5.2A is a

$$\frac{x \notin \mathrm{Dom}(H)}{H; \overline{L}; \mathsf{let}\ \ell, x{=}v;\ s \xrightarrow{\mathrm{s}} H, x \mapsto v; \overline{L}; \cdot; s}\ \mathrm{DS5.1}$$

$$\frac{}{H; \overline{L}; (v; s) \xrightarrow{\mathrm{s}} H; \overline{L}; \cdot; s}\ \mathrm{DS5.2} \qquad \frac{}{H; \overline{L}; (\mathsf{return}\ v; s) \xrightarrow{\mathrm{s}} H; \overline{L}; \cdot; \mathsf{return}\ v}\ \mathrm{DS5.3}$$

$$\frac{}{H; \overline{L}; \mathsf{if}\ 0\ s_1\ s_2 \xrightarrow{\mathrm{s}} H; \overline{L}; \cdot; s_2}\ \mathrm{DS5.4} \qquad \frac{i \neq 0}{H; \overline{L}; \mathsf{if}\ i\ s_1\ s_2 \xrightarrow{\mathrm{s}} H; \overline{L}; \cdot; s_1}\ \mathrm{DS5.5}$$

$$\frac{}{H; \overline{L}; \mathsf{while}\ e\ s \xrightarrow{\mathrm{s}} H; \overline{L}; \cdot; \mathsf{if}\ e\ (s; \mathsf{while}\ e\ s)\ 0}\ \mathrm{DS5.6}$$

$$\frac{}{H; \overline{L}; (\mathsf{open}(\mathsf{pack}\ \tau, v\ \mathsf{as}\ \exists\alpha{:}\kappa[\gamma].\tau')\ \mathsf{as}\ \ell, \alpha, x;\ s) \xrightarrow{\mathrm{s}} H; \overline{L}; \cdot; \mathsf{let}\ \ell, x{=}v;\ s[\tau/\alpha]}\ \mathrm{DS5.7}$$

$$\frac{}{H; (L; L_0, i; L_h); \mathsf{sync\ lock}\ i\ s \xrightarrow{\mathrm{s}} H; (L; L_0; L_h, i); \cdot; (s; \mathsf{release}\ i)}\ \mathrm{DS5.8}$$

$$\frac{}{H; \overline{L}; \mathsf{sync\ nonlock}\ s \xrightarrow{\mathrm{s}} H; \overline{L}; \cdot; s}\ \mathrm{DS5.9}$$

$$\frac{}{H; (L; L_0; L_h, i); (v; \mathsf{release}\ i) \xrightarrow{\mathrm{s}} H; (L; L_0, i; L_h); \cdot; v}\ \mathrm{DS5.10}$$

$$\frac{}{H; (L; L_0; L_h, i); (\mathsf{return}\ v; \mathsf{release}\ i) \xrightarrow{\mathrm{s}} H; (L; L_0, i; L_h); \cdot; \mathsf{return}\ v}\ \mathrm{DS5.11}$$

$$\frac{}{H; \overline{L}; \mathsf{spawn}\ v_1(v_2) \xrightarrow{\mathrm{s}} H; \overline{L}; \mathsf{return}\ v_1(v_2); 0}\ \mathrm{DS5.12}$$

$$\frac{H; \overline{L}; e \xrightarrow{\mathrm{r}} H'; \overline{L}'; s_{\mathrm{opt}}; e'}{\begin{array}{r} H; \overline{L}; e \xrightarrow{\mathrm{s}} H'; \overline{L}'; s_{\mathrm{opt}}; e' \\ H; \overline{L}; \mathsf{return}\ e \xrightarrow{\mathrm{s}} H'; \overline{L}'; s_{\mathrm{opt}}; \mathsf{return}\ e' \\ H; \overline{L}; \mathsf{if}\ e\ s_1\ s_2 \xrightarrow{\mathrm{s}} H'; \overline{L}'; s_{\mathrm{opt}}; \mathsf{if}\ e'\ s_1\ s_2 \\ H; \overline{L}; \mathsf{let}\ \ell, x{=}e;\ s \xrightarrow{\mathrm{s}} H'; \overline{L}'; s_{\mathrm{opt}}; \mathsf{let}\ \ell, x{=}e';\ s \\ H; \overline{L}; \mathsf{open}\ e\ \mathsf{as}\ \ell, \alpha, x;\ s \xrightarrow{\mathrm{s}} H'; \overline{L}'; s_{\mathrm{opt}}; \mathsf{open}\ e'\ \mathsf{as}\ \ell, \alpha, x;\ s \\ H; \overline{L}; \mathsf{sync}\ e\ s \xrightarrow{\mathrm{s}} H'; \overline{L}'; s_{\mathrm{opt}}; \mathsf{sync}\ e'\ s \\ H; \overline{L}; \mathsf{spawn}\ e(e_2) \xrightarrow{\mathrm{s}} H'; \overline{L}'; s_{\mathrm{opt}}; \mathsf{spawn}\ e'(e_2) \\ H; \overline{L}; \mathsf{spawn}\ v(e) \xrightarrow{\mathrm{s}} H'; \overline{L}'; s_{\mathrm{opt}}; \mathsf{spawn}\ v(e') \end{array}}\ \mathrm{DS5.13}$$

$$\frac{H; \overline{L}; s \xrightarrow{\mathrm{s}} H'; \overline{L}'; s_{\mathrm{opt}}; s'}{\begin{array}{r} H; \overline{L}; (s; s_2) \xrightarrow{\mathrm{s}} H'; \overline{L}'; s_{\mathrm{opt}}; (s'; s_2) \\ H; \overline{L}; (s; \mathsf{release}\ i) \xrightarrow{\mathrm{s}} H'; \overline{L}'; s_{\mathrm{opt}}; (s'; \mathsf{release}\ i) \end{array}}\ \mathrm{DS5.14}$$

Figure 5.5: Chapter 5 Dynamic Semantics, Statements

$$\frac{}{H; \overline{L}; x \xrightarrow{\mathrm{r}} H; \overline{L}; \cdot; H(x)}\ \text{DR5.1}$$

$$\frac{}{H, x \mapsto v'; \overline{L}; x{=}v \xrightarrow{\mathrm{r}} H, x \mapsto \mathsf{junk}_v; \overline{L}; \cdot; x{=}\mathsf{junk}_v}\ \text{DR5.2A}$$

$$\frac{}{H, x \mapsto \mathsf{junk}_v; \overline{L}; x{=}\mathsf{junk}_v \xrightarrow{\mathrm{r}} H, x \mapsto v; \overline{L}; \cdot; v}\ \text{DR5.2B}$$

$$\frac{}{H; \overline{L}; *\&x \xrightarrow{\mathrm{r}} H; \overline{L}; \cdot; x}\ \text{DR5.3} \qquad \frac{}{H; \overline{L}; (v_0, v_1).i \xrightarrow{\mathrm{r}} H; \overline{L}; \cdot; v_i}\ \text{DR5.4}$$

$$\frac{}{H; \overline{L}; ((\tau_1, \ell\ x) \xrightarrow{\epsilon} \tau_2\ s)(v) \xrightarrow{\mathrm{r}} H; \overline{L}; \cdot; \mathsf{call}\ (\mathsf{let}\ \ell, x{=}v;\ \ s)}\ \text{DR5.5}$$

$$\frac{}{H; \overline{L}; \mathsf{call}\ \mathsf{return}\ v \xrightarrow{\mathrm{r}} H; \overline{L}; \cdot; v}\ \text{DR5.6} \qquad \frac{}{H; \overline{L}; (\Lambda\alpha{:}\kappa[\gamma].f)[\tau] \xrightarrow{\mathrm{r}} H; \overline{L}; \cdot; f[\tau/\alpha]}\ \text{DR5.7}$$

$$\frac{i \notin L}{\begin{array}{l} H; (L; L_0; L_h); \mathsf{newlock}() \xrightarrow{\mathrm{r}} \\ H; (L, i; L_0, i; L_h); \cdot; \mathsf{pack}\ \mathrm{S}(i), \mathsf{lock}\ i\ \mathsf{as}\ \exists\alpha{:}\mathtt{LS}[\cdot].\mathrm{lock}(\alpha) \end{array}}\ \text{DR5.8}$$

$$\frac{H; \overline{L}; s \xrightarrow{\mathrm{s}} H'; \overline{L}'; s_{\mathrm{opt}}; s'}{H; \overline{L}; \mathsf{call}\ s \xrightarrow{\mathrm{r}} H'; \overline{L}'; s_{\mathrm{opt}}; \mathsf{call}\ s'}\ \text{DR5.9} \qquad \frac{H; \overline{L}; e \xrightarrow{\mathrm{l}} H'; \overline{L}'; s_{\mathrm{opt}}; e'}{\begin{array}{c} H; \overline{L}; \&e \xrightarrow{\mathrm{r}} H'; \overline{L}'; s_{\mathrm{opt}}; \&e' \\ H; \overline{L}; e{=}e_2 \xrightarrow{\mathrm{r}} H'; \overline{L}'; s_{\mathrm{opt}}; e'{=}e_2 \end{array}}\ \text{DR5.10}$$

$$\frac{H; \overline{L}; e \xrightarrow{\mathrm{r}} H'; \overline{L}'; s_{\mathrm{opt}}; e'}{\begin{array}{cc} H; \overline{L}; *e \xrightarrow{\mathrm{r}} H'; \overline{L}'; s_{\mathrm{opt}}; *e' & H; \overline{L}; (e, e_2) \xrightarrow{\mathrm{r}} H'; \overline{L}'; s_{\mathrm{opt}}; (e', e_2) \\ H; \overline{L}; e.i \xrightarrow{\mathrm{r}} H'; \overline{L}'; s_{\mathrm{opt}}; e'.i & H; \overline{L}; (v, e) \xrightarrow{\mathrm{r}} H'; \overline{L}'; s_{\mathrm{opt}}; (v, e') \\ H; \overline{L}; x{=}e \xrightarrow{\mathrm{r}} H'; \overline{L}'; s_{\mathrm{opt}}; x{=}e' & H; \overline{L}; e(e_2) \xrightarrow{\mathrm{r}} H'; \overline{L}'; s_{\mathrm{opt}}; e'(e_2) \\ H; \overline{L}; e[\tau] \xrightarrow{\mathrm{r}} H'; \overline{L}'; s_{\mathrm{opt}}; e'[\tau] & H; \overline{L}; v(e) \xrightarrow{\mathrm{r}} H'; \overline{L}'; s_{\mathrm{opt}}; v(e') \\ \multicolumn{2}{c}{H; \overline{L}; \mathsf{pack}\ \tau', e\ \mathsf{as}\ \tau \xrightarrow{\mathrm{r}} H'; \overline{L}'; s_{\mathrm{opt}}; \mathsf{pack}\ \tau', e'\ \mathsf{as}\ \tau} \end{array}}\ \text{DR5.11}$$

$$\frac{}{H; \overline{L}; *\&x \xrightarrow{\mathrm{l}} H; \overline{L}; \cdot; x}\ \text{DL5.1} \qquad \frac{H; \overline{L}; e \xrightarrow{\mathrm{r}} H'; \overline{L}'; s_{\mathrm{opt}}; e'}{H; \overline{L}; *e \xrightarrow{\mathrm{l}} H'; \overline{L}'; s_{\mathrm{opt}}; e'}\ \text{DL5.2}$$

Figure 5.6: Chapter 5 Dynamic Semantics, Expressions

$$
\begin{array}{rrcl}
\text{constraint locks:} & \text{locks}(\cdot) & = & \emptyset \\
& \text{locks}(\gamma, \epsilon_1 \subseteq \epsilon_2) & = & \text{locks}(\gamma) \cup \epsilon_1 \cup \epsilon_2 \\
\text{type locks:} & \text{locks}(\alpha) & = & \alpha \\
& \text{locks}(\text{int}) & = & \emptyset \\
& \text{locks}(\tau_0 \times \tau_1) & = & \text{locks}(\tau_0) \cup \text{locks}(\tau_1) \\
& \text{locks}(\tau_1 \xrightarrow{\epsilon} \tau_2) & = & \epsilon \\
& \text{locks}(\tau * \ell) & = & \text{locks}(\tau) \cup \text{locks}(\ell) \\
& \text{locks}(\forall \alpha{:}\kappa[\gamma].\tau) & = & (\text{locks}(\gamma) \cup \text{locks}(\tau)) - \alpha \\
& \text{locks}(\exists \alpha{:}\kappa[\gamma].\tau) & = & (\text{locks}(\gamma) \cup \text{locks}(\tau)) - \alpha \\
& \text{locks}(\text{lock}(\ell)) & = & \emptyset \\
& \text{locks}(\text{S}(i)) & = & i \\
& \text{locks}(loc) & = & \emptyset
\end{array}
$$

Notes: We omit the formal definition of substitution because it is almost identical to the Chapter 4 definition (Figure 4.5). The changes are: (1) $\alpha[\tau/\alpha] = \text{locks}(\tau)$ where α is an effect, (2) $\text{lock}(\ell)[\tau/\alpha] = \text{lock}(\ell[\tau/\alpha])$, and (3) $loc[\tau/\alpha] = loc$.

Figure 5.7: Chapter 5 Dynamic Semantics, Type Substitution

state in which DR5.2B applies, but the machine might evaluate other threads in-between. Note that DR5.2A does not apply if the location x holds some $\mathsf{junk}_{v'}$. If we relaxed the rule in this way, a write-write data race could go undetected. With this precondition, the type-safety theorem in the next section precludes write-write races because it implies that a thread cannot be stuck because x holds some $\mathsf{junk}_{v'}$. Rule DR5.8 creates a new lock. It uses L to ensure the new lock is uniquely identified. The result is an existential package with the same type as $\mathsf{newlock}()$.

Rules DS5.7 and DR5.7 use substitution to eliminate type variables. Figure 5.7 describes the definition of substitution. The interesting part is replacing α with $\text{locks}(\tau)$ in effects when substituting τ for α. The definition of $\text{locks}(\tau)$ is almost all free lock names in τ (omitting loc), just as $\text{regions}(\tau)$ in Chapter 4 is all free region names in τ. However, $\text{locks}(\text{lock}(\ell)) = \emptyset$. We do not need to hold a lock to acquire it (in fact we should *not* hold it), so choosing $\text{locks}(\text{lock}(\ell)) = \text{locks}(\ell)$ is a less sensible choice. Nonetheless, any definition of $\text{locks}(\tau)$ that does not introduce free type variables is safe.

It is straightforward to check that types have no essential run-time effect. We do not prove a type-erasure result, but we expect doing so is straightforward.

5.6.3 Static Semantics

A valid source program is a statement (leading to a program state of the form $\cdot;\cdot;\cdot;(\cdot;s)$) that type-checks under an empty context ($\cdot;\cdot;\cdot;\cdot;\emptyset;\tau \vdash_{\text{styp}} s$), does not terminate without returning ($\vdash_{\text{ret}} s$), does not contain any release statements ($\cdot \vdash_{\text{srel}} s$), and does not contain any junk expressions ($\vdash_{\text{jf}} s$).

Many judgments are very similar to those in Chapter 4. Three interdependent judgments define type-checking for statements, right-expressions, and left-expressions (Figures 5.10 and 5.11). Expressions that access memory type-check only if the current capability and constraints establish that the thread holds the lock that guards the location or the location is thread-local. We use the judgments $\gamma;\epsilon \vdash_{\text{acc}} \ell$, $\gamma \vdash_{\text{eff}} \epsilon_1 \subseteq \epsilon_2$, and $\gamma \vdash_{\text{eff}} \gamma'$ (Figure 5.9) to ensure threads hold the necessary locks to access memory, call functions, eliminate universal types, and introduce existential types.

The judgments in Figure 5.8 define various properties of type-level and kind-level constructs. The $\vdash_{\text{k}}$ and $\vdash_{\text{wf}}$ judgments ensure types have the correct kinds and there are no references to free type variables. Kinding has a subsumption rule for the subkinding that $\vdash_{\text{sk}}$ defines. The $\vdash_{\text{shr}}$ and $\vdash_{\text{loc}}$ judgments help partition a heap's type into shared and local portions. If $L \vdash_{\text{shr}} \Gamma$, then every location in Γ is sharable (its type and lock name have sharable kind). If $L \vdash_{\text{loc}} \Gamma$, then no location in Γ is sharable. Assuming $L;\cdot \vdash_{\text{wf}} \Gamma$, there are unique Γ_1 and Γ_2 such that $\Gamma = \Gamma_1\Gamma_2$, $L \vdash_{\text{shr}} \Gamma_1$, and $L \vdash_{\text{loc}} \Gamma_2$.

For nonsource programs, we must relax the ban on release statements and junk expressions. For the former, the judgments $\vdash_{\text{srel}}$ and $\vdash_{\text{erel}}$ (Figure 5.13) work like $\vdash_{\text{spop}}$ and $\vdash_{\text{epop}}$ in Chapter 4. More specifically, $L \vdash_{\text{srel}} s$ ensures s releases only locks from L, releases no lock more than once, and does not need to hold some lock after the lock is released. For the former, we use the judgments in Figure 5.14, which formalize the intuition that junk should only appear if a thread is in the process of mutating a heap location. More specifically, $\vdash_{\text{j}} H;s$ if either H and s are junk-free or else $H = H'x \mapsto \mathsf{junk}_v$, H' is junk-free, and s is junk-free except that its "active redex" is x=junk_v.

The judgments in Figure 5.15 type-check heaps and program states. The $\vdash_{\text{htyp}}$ ensures heap values have appropriate types and consistent assumptions about what locks guard what locations. We use $\vdash_{\text{hlk}}$ to partition the heap according to the locks that different threads hold. Finally, $\vdash_{\text{prog}}$ partitions the heap appropriately and ensures the entire state is well-formed. More specifically, L should describe exactly the locks that are available or held by some thread, and none of the other lock sets should share elements. Given the one heap H, we can divide it into a shared heap H_S and thread-local heaps $H_{1U}, \ldots, H_{nU}$. The shared heap is closed and well-typed. The thread-local heaps are well-typed, but each may refer to

$$\frac{}{L;\Delta \vdash_{\text{wf}} \emptyset} \qquad \frac{\alpha \in \text{Dom}(\Delta)}{L;\Delta \vdash_{\text{wf}} \alpha} \qquad \frac{i \in L}{L;\Delta \vdash_{\text{wf}} i} \qquad \frac{L;\Delta \vdash_{\text{wf}} \epsilon_1 \quad L;\Delta \vdash_{\text{wf}} \epsilon_2}{L;\Delta \vdash_{\text{wf}} \epsilon_1 \cup \epsilon_2}$$

$$\frac{}{L;\Delta \vdash_{\text{wf}} \cdot} \qquad \frac{L;\Delta \vdash_{\text{wf}} \gamma \quad L;\Delta \vdash_{\text{wf}} \epsilon_1 \quad L;\Delta \vdash_{\text{wf}} \epsilon_2}{L;\Delta \vdash_{\text{wf}} \gamma,\ \epsilon_1 \subseteq \epsilon_2}$$

$$\frac{}{\vdash_{\text{sk}} \mathtt{B}\sigma \leq \mathtt{A}\sigma} \qquad \frac{}{\vdash_{\text{sk}} \theta\mathtt{S} \leq \theta\mathtt{U}} \qquad \frac{\vdash_{\text{sk}} \kappa_1 \leq \kappa_3 \quad \vdash_{\text{sk}} \kappa_3 \leq \kappa_2}{\vdash_{\text{sk}} \kappa_1 \leq \kappa_2} \qquad \frac{L;\Delta \vdash_{\text{k}} \tau : \kappa' \quad \vdash_{\text{sk}} \kappa' \leq \kappa}{L;\Delta \vdash_{\text{k}} \tau : \kappa}$$

$$\frac{}{L;\Delta \vdash_{\text{k}} \text{int} : \mathtt{BS}} \qquad \frac{\alpha \in \text{Dom}(\Delta)}{L;\Delta \vdash_{\text{k}} \alpha : \Delta(\alpha)} \qquad \frac{i \in L}{L;\Delta \vdash_{\text{k}} \text{S}(i) : \mathtt{LS}} \qquad \frac{}{L;\Delta \vdash_{\text{k}} loc : \mathtt{LU}}$$

$$\frac{L;\Delta \vdash_{\text{k}} \tau_0 : \mathtt{A}\sigma \quad L;\Delta \vdash_{\text{k}} \tau_1 : \mathtt{A}\sigma}{L;\Delta \vdash_{\text{k}} \tau_0 \times \tau_1 : \mathtt{A}\sigma} \qquad \frac{L;\Delta \vdash_{\text{k}} \tau_1 : \mathtt{AU} \quad L;\Delta \vdash_{\text{k}} \tau_2 : \mathtt{AU} \quad L;\Delta \vdash_{\text{wf}} \epsilon}{L;\Delta \vdash_{\text{k}} \tau_1 \xrightarrow{\epsilon} \tau_2 : \mathtt{AS}}$$

$$\frac{L;\Delta \vdash_{\text{k}} \tau : \mathtt{A}\sigma \quad L;\Delta \vdash_{\text{k}} \ell : \mathtt{L}\sigma}{L;\Delta \vdash_{\text{k}} \tau * \ell : \mathtt{B}\sigma} \qquad \frac{L;\Delta \vdash_{\text{k}} \ell : \mathtt{L}\sigma}{L;\Delta \vdash_{\text{k}} \text{lock}(\ell) : \mathtt{A}\sigma}$$

$$\frac{L;\Delta, \alpha{:}\theta\sigma \vdash_{\text{k}} \tau : \mathtt{A}\sigma' \quad L;\Delta, \alpha{:}\theta\sigma \vdash_{\text{wf}} \gamma \quad \theta \neq \mathtt{A} \quad \alpha \notin \text{Dom}(\Delta)}{\begin{array}{c} L;\Delta \vdash_{\text{k}} \forall \alpha{:}\theta\sigma[\gamma].\tau : \mathtt{A}\sigma' \\ L;\Delta \vdash_{\text{k}} \exists \alpha{:}\theta\sigma[\gamma].\tau : \mathtt{A}\sigma' \end{array}}$$

$$\frac{}{L;\Delta \vdash_{\text{wf}} \cdot} \qquad \frac{L;\Delta \vdash_{\text{wf}} \Gamma \quad L;\Delta \vdash_{\text{k}} \tau : \mathtt{AU} \quad L;\Delta \vdash_{\text{k}} \ell : \mathtt{LU}}{L;\Delta \vdash_{\text{wf}} \Gamma, x{:}(\tau, \ell)}$$

$$\frac{L;\Delta \vdash_{\text{wf}} \Gamma \quad L;\Delta \vdash_{\text{wf}} \gamma \quad L;\Delta \vdash_{\text{wf}} \epsilon}{\vdash_{\text{wf}} L;\Delta;\Gamma;\gamma;\epsilon}$$

$$\frac{}{L \vdash_{\text{shr}} \cdot} \qquad \frac{L \vdash_{\text{shr}} \Gamma \quad L;\cdot \vdash_{\text{k}} \tau : \mathtt{AS} \quad L;\cdot \vdash_{\text{k}} \ell : \mathtt{LS}}{L \vdash_{\text{shr}} \Gamma, x{:}(\tau, \ell)}$$

$$\frac{}{L \vdash_{\text{loc}} \cdot} \qquad \frac{L \vdash_{\text{loc}} \Gamma \quad L;\cdot \not\vdash_{\text{k}} \tau : \mathtt{AS}}{L \vdash_{\text{loc}} \Gamma, x{:}(\tau, \ell)} \qquad \frac{L \vdash_{\text{loc}} \Gamma \quad L;\cdot \not\vdash_{\text{k}} \ell : \mathtt{LS}}{L \vdash_{\text{loc}} \Gamma, x{:}(\tau, \ell)}$$

Figure 5.8: Chapter 5 Kinding, Well-Formedness, and Context Sharability

$$\frac{}{\gamma;\epsilon \vdash_{\text{acc}} loc} \qquad \frac{\gamma \vdash_{\text{eff}} i \subseteq \epsilon}{\gamma;\epsilon \vdash_{\text{acc}} \mathrm{S}(i)} \qquad \frac{\gamma \vdash_{\text{eff}} \alpha \subseteq \epsilon}{\gamma;\epsilon \vdash_{\text{acc}} \alpha}$$

$$\frac{}{\gamma \vdash_{\text{eff}} \epsilon \subseteq \epsilon} \qquad \frac{}{\gamma_1, \epsilon_1 \subseteq \epsilon_2, \gamma_2 \vdash_{\text{eff}} \epsilon_1 \subseteq \epsilon_2}$$

$$\frac{\gamma \vdash_{\text{eff}} \epsilon_1 \subseteq \epsilon \quad \gamma \vdash_{\text{eff}} \epsilon_2 \subseteq \epsilon}{\gamma \vdash_{\text{eff}} \epsilon_1 \cup \epsilon_2 \subseteq \epsilon} \qquad \frac{\gamma \vdash_{\text{eff}} \epsilon \subseteq \epsilon_1}{\gamma \vdash_{\text{eff}} \epsilon \subseteq \epsilon_1 \cup \epsilon_2} \qquad \frac{\gamma \vdash_{\text{eff}} \epsilon_1 \subseteq \epsilon_3 \quad \gamma \vdash_{\text{eff}} \epsilon_3 \subseteq \epsilon_2}{\gamma \vdash_{\text{eff}} \epsilon_1 \subseteq \epsilon_2}$$

$$\frac{}{\gamma \vdash_{\text{eff}} .} \qquad \frac{\gamma \vdash_{\text{eff}} \gamma' \quad \gamma \vdash_{\text{eff}} \epsilon_1 \subseteq \epsilon_2}{\gamma \vdash_{\text{eff}} \gamma', \epsilon_1 \subseteq \epsilon_2}$$

Figure 5.9: Chapter 5 Effect and Constraint Containment

$$\frac{C \vdash_{\text{rtyp}} e : \tau'}{C;\tau \vdash_{\text{styp}} e}\ \text{SS5.1} \qquad \frac{C \vdash_{\text{rtyp}} e : \tau}{C;\tau \vdash_{\text{styp}} \mathsf{return}\ e}\ \text{SS5.2} \qquad \frac{C;\tau \vdash_{\text{styp}} s_1 \quad C;\tau \vdash_{\text{styp}} s_2}{C;\tau \vdash_{\text{styp}} s_1; s_2}\ \text{SS5.3}$$

$$\frac{C \vdash_{\text{rtyp}} e : \text{int} \quad C;\tau \vdash_{\text{styp}} s}{C;\tau \vdash_{\text{styp}} \mathsf{while}\ e\ s}\ \text{SS5.4} \qquad \frac{C \vdash_{\text{rtyp}} e : \text{int} \quad C;\tau \vdash_{\text{styp}} s_1 \quad C;\tau \vdash_{\text{styp}} s_2}{C;\tau \vdash_{\text{styp}} \mathsf{if}\ e\ s_1\ s_2}\ \text{SS5.5}$$

$$\frac{L;\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rtyp}} e : \tau' \quad L;\Delta;\Gamma, x{:}(\tau',\ell);\gamma;\epsilon;\tau \vdash_{\text{styp}} s \quad x \notin \mathrm{Dom}(\Gamma)}{L;\Delta;\Gamma;\gamma;\epsilon;\tau \vdash_{\text{styp}} \mathsf{let}\ \ell, x{=}e;\ \ s}\ \text{SS5.6}$$

$$\frac{\begin{array}{l} L;\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rtyp}} e : \exists\alpha{:}\kappa[\gamma'].\tau' \qquad \alpha \notin \mathrm{Dom}(\Delta) \qquad x \notin \mathrm{Dom}(\Gamma) \\ L;\Delta, \alpha{:}\kappa;\Gamma, x{:}(\tau',\ell);\gamma\gamma';\epsilon;\tau \vdash_{\text{styp}} s \qquad L;\Delta \vdash_{\text{k}} \ell : \mathtt{LU} \qquad L;\Delta \vdash_{\text{k}} \tau : \mathtt{AU} \end{array}}{L;\Delta;\Gamma;\gamma;\epsilon;\tau \vdash_{\text{styp}} \mathsf{open}\ e\ \mathsf{as}\ \ell, \alpha, x;\ \ s}\ \text{SS5.7}$$

$$\frac{L;\Delta;\Gamma;\gamma;\epsilon \vdash_{\text{rtyp}} e : \mathrm{lock}(\ell) \quad L;\Delta;\Gamma;\gamma;\epsilon \cup \mathrm{locks}(\ell);\tau \vdash_{\text{styp}} s}{L;\Delta;\Gamma;\gamma;\epsilon;\tau \vdash_{\text{styp}} \mathsf{sync}\ e\ s}\ \text{SS5.8}$$

$$\frac{L;\Delta;\Gamma;\gamma;\epsilon \cup i;\tau \vdash_{\text{styp}} s}{L;\Delta;\Gamma;\gamma;\epsilon;\tau \vdash_{\text{styp}} s; \mathsf{release}\ i}\ \text{SS5.9}$$

$$\frac{C \vdash_{\text{rtyp}} e_1 : \tau_1 \xrightarrow{\emptyset} \tau_2 \quad C \vdash_{\text{rtyp}} e_2 : \tau_1 \quad C_L; C_\Delta \vdash_{\text{k}} \tau_1 : \mathtt{AS}}{C;\tau \vdash_{\text{styp}} \mathsf{spawn}\ e_1(e_2)}\ \text{SS5.10}$$

Figure 5.10: Chapter 5 Typing, Statements

$$\frac{C_\Gamma(x) = \tau, \ell \quad \vdash_{\text{wf}} C}{C \vdash_{\text{ltyp}} x : \tau, \ell}\ \text{SL5.1} \qquad \frac{C \vdash_{\text{rtyp}} e : \tau{*}\ell}{C \vdash_{\text{ltyp}} {*}e : \tau, \ell}\ \text{SL5.2}$$

$$\frac{C_\Gamma(x) = \tau, \ell \quad C_\gamma; C_\epsilon \vdash_{\text{acc}} \ell \quad \vdash_{\text{wf}} C}{C \vdash_{\text{rtyp}} x : \tau}\ \text{SR5.1} \qquad \frac{C \vdash_{\text{rtyp}} e : \tau{*}\ell \quad C_\gamma; C_\epsilon \vdash_{\text{acc}} \ell}{C \vdash_{\text{rtyp}} {*}e : \tau}\ \text{SR5.2}$$

$$\frac{C \vdash_{\text{rtyp}} e : \tau_0 \times \tau_1}{C \vdash_{\text{rtyp}} e.0 : \tau_0}\ \text{SR5.3} \qquad \frac{C \vdash_{\text{rtyp}} e : \tau_0 \times \tau_1}{C \vdash_{\text{rtyp}} e.1 : \tau_1}\ \text{SR5.4} \qquad \frac{\vdash_{\text{wf}} C}{C \vdash_{\text{rtyp}} i : \text{int}}\ \text{SR5.5}$$

$$\frac{C \vdash_{\text{ltyp}} e : \tau, \ell}{C \vdash_{\text{rtyp}} \&e : \tau{*}\ell}\ \text{SR5.6} \qquad \frac{C \vdash_{\text{rtyp}} e_0 : \tau_0 \quad C \vdash_{\text{rtyp}} e_1 : \tau_1}{C \vdash_{\text{rtyp}} (e_0, e_1) : \tau_0 \times \tau_1}\ \text{SR5.7}$$

$$\frac{C \vdash_{\text{ltyp}} e_1 : \tau, \ell \quad C \vdash_{\text{rtyp}} e_2 : \tau \quad C_\gamma; C_\epsilon \vdash_{\text{acc}} \ell}{C \vdash_{\text{rtyp}} e_1{=}e_2 : \tau}\ \text{SR5.8}$$

$$\frac{C \vdash_{\text{rtyp}} e_1 : \tau' \xrightarrow{\epsilon'} \tau \quad C \vdash_{\text{rtyp}} e_2 : \tau' \quad C_\gamma \vdash_{\text{eff}} \epsilon' \subseteq C_\epsilon}{C \vdash_{\text{rtyp}} e_1(e_2) : \tau}\ \text{SR5.9} \qquad \frac{C; \tau \vdash_{\text{styp}} s \quad \vdash_{\text{ret}} s}{C \vdash_{\text{rtyp}} \mathsf{call}\ s : \tau}\ \text{SR5.10}$$

$$\frac{C \vdash_{\text{rtyp}} e : \forall \alpha{:}\kappa[\gamma'].\tau' \quad C_L; C_\Delta \vdash_{\text{k}} \tau : \kappa \quad C_\gamma \vdash_{\text{eff}} \gamma'[\tau/\alpha]}{C \vdash_{\text{rtyp}} e[\tau] : \tau'[\tau/\alpha]}\ \text{SR5.11}$$

$$\frac{C \vdash_{\text{rtyp}} e : \tau[\tau'/\alpha] \quad C_L; C_\Delta \vdash_{\text{k}} \tau' : \kappa \quad C_\gamma \vdash_{\text{eff}} \gamma'[\tau'/\alpha] \quad C_L; C_\Delta \vdash_{\text{k}} \exists \alpha{:}\kappa[\gamma'].\tau : \mathtt{AU}}{C \vdash_{\text{rtyp}} \mathsf{pack}\ \tau', e\ \mathsf{as}\ \exists \alpha{:}\kappa[\gamma'].\tau : \exists \alpha{:}\kappa[\gamma'].\tau}\ \text{SR5.12}$$

$$\frac{\begin{array}{c} L; \Delta; \Gamma_1, x{:}(\tau_1, \ell); \gamma; \epsilon'; \tau_2 \vdash_{\text{styp}} s \qquad \vdash_{\text{ret}} s \qquad x \notin \text{Dom}(\Gamma) \\ \Gamma = \Gamma_1\Gamma_2 \qquad L \vdash_{\text{shr}} \Gamma_1 \qquad L; \cdot \vdash_{\text{wf}} \Gamma_1 \qquad \vdash_{\text{wf}} L; \Delta; \Gamma; \gamma; \epsilon \end{array}}{L; \Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} (\tau_1, \ell\ x) \xrightarrow{\epsilon'} \tau_2\ s : \tau_1 \xrightarrow{\epsilon'} \tau_2}\ \text{SR5.13}$$

$$\frac{L; \Delta, \alpha{:}\kappa; \Gamma; \gamma\gamma'; \epsilon \vdash_{\text{rtyp}} f : \tau \quad \vdash_{\text{wf}} L; \Delta; \Gamma; \gamma; \epsilon \quad L; \Delta \vdash_{\text{k}} \forall \alpha{:}\kappa[\gamma'].\tau : \mathtt{AU}}{L; \Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} \Lambda \alpha{:}\kappa[\gamma'].f : \forall \alpha{:}\kappa[\gamma'].\tau}\ \text{SR5.14}$$

$$\frac{i \in C_L \quad \vdash_{\text{wf}} C}{C \vdash_{\text{rtyp}} \mathsf{lock}\ i : \text{lock}(\text{S}(i))}\ \text{SR5.15} \qquad \frac{C \vdash_{\text{rtyp}} v : \tau}{C \vdash_{\text{rtyp}} \mathsf{junk}_v : \tau}\ \text{SR5.16}$$

$$\frac{\vdash_{\text{wf}} C}{C \vdash_{\text{rtyp}} \mathsf{nonlock} : \text{lock}(loc)}\ \text{SR5.17} \qquad \frac{\vdash_{\text{wf}} C}{C \vdash_{\text{rtyp}} \mathsf{newlock}() : \exists \alpha{:}\mathtt{LS}[\cdot].\text{lock}(\alpha)}\ \text{SR5.18}$$

Figure 5.11: Chapter 5 Typing, Expressions

$$\frac{}{\vdash_{\text{ret}} \mathsf{return}\ e} \qquad \frac{\vdash_{\text{ret}} s_1 \quad \vdash_{\text{ret}} s_2}{\vdash_{\text{ret}} \mathsf{if}\ e\ s_1\ s_2}$$

$$\frac{\vdash_{\text{ret}} s}{\begin{array}{ll} \vdash_{\text{ret}} s; s' & \vdash_{\text{ret}} \mathsf{let}\ \ell, x{=}e;\ s \\ \vdash_{\text{ret}} s'; s & \vdash_{\text{ret}} \mathsf{open}\ e\ \mathsf{as}\ \ell, \alpha, x;\ s \\ \vdash_{\text{ret}} s; \mathsf{release}\ i & \vdash_{\text{ret}} \mathsf{sync}\ e\ s \end{array}}$$

Figure 5.12: Chapter 5 Must-Return

$$\frac{L \vdash_{\text{srel}} s}{i, L \vdash_{\text{srel}} s; \mathsf{release}\ i} \qquad \frac{L \vdash_{\text{srel}} s_1 \quad \cdot \vdash_{\text{srel}} s_2}{L \vdash_{\text{srel}} s_1; s_2} \qquad \frac{L \vdash_{\text{erel}} e \quad \cdot \vdash_{\text{srel}} s_1 \quad \cdot \vdash_{\text{srel}} s_2}{L \vdash_{\text{srel}} \mathsf{if}\ e\ s_1\ s_2} \qquad \frac{\cdot \vdash_{\text{erel}} e \quad \cdot \vdash_{\text{erel}} s}{\cdot \vdash_{\text{srel}} \mathsf{while}\ e\ s}$$

$$\frac{L \vdash_{\text{erel}} e \quad \cdot \vdash_{\text{srel}} s}{\begin{array}{l} L \vdash_{\text{srel}} \mathsf{let}\ \ell, x{=}e;\ s \\ L \vdash_{\text{srel}} \mathsf{open}\ e\ \mathsf{as}\ \ell, \alpha, x;\ s \\ L \vdash_{\text{srel}} \mathsf{sync}\ e\ s \end{array}} \qquad \frac{L \vdash_{\text{erel}} e}{\begin{array}{l} L \vdash_{\text{srel}} e \\ L \vdash_{\text{srel}} \mathsf{return}\ e \end{array}} \qquad \frac{L \vdash_{\text{erel}} e_1 \quad \cdot \vdash_{\text{erel}} e_2}{L \vdash_{\text{srel}} \mathsf{spawn}\ e_1(e_2)} \qquad \frac{\cdot \vdash_{\text{erel}} v \quad L \vdash_{\text{erel}} e}{L \vdash_{\text{srel}} \mathsf{spawn}\ v(e)}$$

$$\frac{}{\begin{array}{l} \cdot \vdash_{\text{erel}} x \\ \cdot \vdash_{\text{erel}} i \\ \cdot \vdash_{\text{erel}} \mathsf{lock}\ i \\ \cdot \vdash_{\text{erel}} \mathsf{nonlock} \\ \cdot \vdash_{\text{erel}} \mathsf{newlock}() \end{array}} \qquad \frac{L \vdash_{\text{erel}} e}{\begin{array}{l} L \vdash_{\text{erel}} \&e \\ L \vdash_{\text{erel}} *e \\ L \vdash_{\text{erel}} e.i \\ L \vdash_{\text{erel}} x{=}e \\ L \vdash_{\text{erel}} e[\tau] \\ L \vdash_{\text{erel}} \mathsf{pack}\ \tau, e\ \mathsf{as}\ \tau' \end{array}} \qquad \frac{L \vdash_{\text{erel}} e_0 \quad \cdot \vdash_{\text{erel}} e_1}{\begin{array}{l} L \vdash_{\text{erel}} (e_0, e_1) \\ L \vdash_{\text{erel}} e_0{=}e_1 \\ L \vdash_{\text{erel}} e_0(e_1) \end{array}} \qquad \frac{\cdot \vdash_{\text{erel}} v \quad L \vdash_{\text{erel}} e}{\begin{array}{l} L \vdash_{\text{erel}} (v, e) \\ L \vdash_{\text{erel}} v(e) \end{array}}$$

$$\frac{\cdot \vdash_{\text{srel}} s}{\cdot \vdash_{\text{erel}} (\tau, \ell\ x) \xrightarrow{\epsilon} \tau'\ s} \qquad \frac{\cdot \vdash_{\text{erel}} f}{\cdot \vdash_{\text{erel}} \Lambda\alpha{:}\kappa[\gamma].f} \qquad \frac{L \vdash_{\text{srel}} s}{L \vdash_{\text{erel}} \mathsf{call}\ s} \qquad \frac{\cdot \vdash_{\text{erel}} v}{\cdot \vdash_{\text{erel}} \mathsf{junk}_v}$$

Figure 5.13: Chapter 5 Typing, Release

$$\frac{x, v \vdash_{\text{js}} s}{x, v \vdash_{\text{js}} s; \mathsf{release}\ i} \qquad \frac{x, v \vdash_{\text{js}} s_1 \quad \vdash_{\text{jf}} s_2}{x, v \vdash_{\text{js}} s_1; s_2} \qquad \frac{x, v \vdash_{\text{je}} e \quad \vdash_{\text{jf}} s_1 \quad \vdash_{\text{jf}} s_2}{x, v \vdash_{\text{js}} \mathsf{if}\ e\ s_1\ s_2}$$

$$\frac{x, v \vdash_{\text{je}} e}{\begin{array}{l} x, v \vdash_{\text{js}} e \\ x, v \vdash_{\text{js}} \mathsf{return}\ e \end{array}} \qquad \frac{x, v \vdash_{\text{je}} e \quad \vdash_{\text{jf}} s}{\begin{array}{l} x, v \vdash_{\text{js}} \mathsf{let}\ \ell, x{=}e;\ s \\ x, v \vdash_{\text{js}} \mathsf{open}\ e\ \mathsf{as}\ \ell, \alpha, x;\ s \\ x, v \vdash_{\text{js}} \mathsf{sync}\ e\ s \end{array}} \qquad \frac{x, v \vdash_{\text{je}} e_1 \quad \vdash_{\text{jf}} e_2}{x, v \vdash_{\text{js}} \mathsf{spawn}\ e_1(e_2)}$$

$$\frac{\vdash_{\text{jf}} v' \quad x, v \vdash_{\text{je}} e_2}{x, v \vdash_{\text{js}} \mathsf{spawn}\ v'(e_2)}$$

$$\frac{\vdash_{\text{jf}} v}{x, v \vdash_{\text{je}} x{=}\mathsf{junk}_v} \qquad \frac{x, v \vdash_{\text{js}} s}{x, v \vdash_{\text{je}} \mathsf{call}\ s}$$

$$\frac{x, v \vdash_{\text{je}} e}{\begin{array}{l} x, v \vdash_{\text{je}} \&e \\ x, v \vdash_{\text{je}} *e \\ x, v \vdash_{\text{je}} e.i \\ x, v \vdash_{\text{je}} x{=}e \\ x, v \vdash_{\text{je}} e[\tau] \\ x, v \vdash_{\text{je}} \mathsf{pack}\ \tau, e\ \mathsf{as}\ \tau' \end{array}}$$

$$\frac{x, v \vdash_{\text{je}} e_1 \quad \vdash_{\text{jf}} e_2}{\begin{array}{l} x, v \vdash_{\text{je}} (e_1, e_2) \\ x, v \vdash_{\text{je}} e_1{=}e_2 \\ x, v \vdash_{\text{je}} e_1(e_2) \end{array}} \qquad \frac{\vdash_{\text{jf}} v' \quad x, v \vdash_{\text{je}} e}{\begin{array}{l} x, v \vdash_{\text{je}} (v', e) \\ x, v \vdash_{\text{je}} v'(e) \end{array}}$$

$$\frac{\vdash_{\text{jf}} H \quad \vdash_{\text{jf}} s}{\vdash_{\text{j}} H; s} \qquad \frac{\vdash_{\text{jf}} H \quad x, v \vdash_{\text{j}} s}{\vdash_{\text{j}} H, x \mapsto \mathsf{junk}_v; s} \qquad \frac{\vdash_{\text{jf}} H \quad \vdash_{\text{jf}} e}{\vdash_{\text{j}} H; e} \qquad \frac{\vdash_{\text{jf}} H \quad x, v \vdash_{\text{je}} e}{\vdash_{\text{j}} H, x \mapsto \mathsf{junk}_v; e}$$

Note: We omit the formal definition of $\vdash_{\text{jf}} e$ (respectively $\vdash_{\text{jf}} s$ and $\vdash_{\text{jf}} H$), which means that no term in e, (respectively s and H) has the form junk_v.

Figure 5.14: Chapter 5 Typing, Junk

$$\frac{}{L;\Gamma \vdash_{\text{htyp}} \cdot : \cdot} \qquad \frac{L;\Gamma \vdash_{\text{htyp}} H : \Gamma' \quad L;\cdot;\Gamma;\cdot;\emptyset \vdash_{\text{rtyp}} e : \tau \quad \cdot \vdash_{\text{erel}} e \quad i \in L}{L;\Gamma \vdash_{\text{htyp}} H, x \mapsto e : \Gamma', x{:}(\tau, S(i))}$$

$$\frac{}{\Gamma; L \vdash_{\text{hlk}} \cdot} \qquad \frac{\Gamma; L \vdash_{\text{hlk}} H \quad \Gamma(x) = (\tau, S(i)) \quad i \in L}{\Gamma; L \vdash_{\text{hlk}} H, x \mapsto e}$$

$$\frac{\begin{array}{l} L = L_0 L_1 \cdots L_n \\ H = H_S H_{1U} \cdots H_{nU} \\ H_S = H_{0S} H_{1S} \cdots H_{nS} \\ L;\Gamma_S \vdash_{\text{htyp}} H_S : \Gamma_S \qquad L \vdash_{\text{shr}} \Gamma_S \\ \Gamma_S; L_0 \vdash_{\text{hlk}} H_{0S} \qquad \vdash_{\text{jf}} H_{0S} \\ \text{for all } 1 \le i \le n \\ \quad L;\Gamma_S\Gamma_{iU} \vdash_{\text{htyp}} H_{iU} : \Gamma_{iU} \qquad L \vdash_{\text{loc}} \Gamma_{iU} \qquad \Gamma_S; L_i \vdash_{\text{hlk}} H_{iS} \\ \quad L;\cdot;\Gamma_S\Gamma_{iU};\cdot;\emptyset;\tau_i \vdash_{\text{styp}} s_i \qquad \vdash_{\text{ret}} s_i \qquad L_i \vdash_{\text{srel}} s_i \qquad \vdash_{\text{j}} H_{iS}H_{iU}; s_i \end{array}}{\vdash_{\text{prog}} L; L_0; H; (L_1; s_1) \cdots (L_n; s_n)}$$

Figure 5.15: Chapter 5 Typing, States

locations in H_S. We can further divide H_S into $H_{0S}H_{1S}H_{nS}$ where H_{0S} holds locations guarded by available locks and the other H_{iS} hold locations guarded by locks that thread i holds. Given all this structure on the heap, the statement s_i should type-check without reference to other threads' local heaps, should return, should release exactly the locks L_i, and should be junk-free except for possibly mutating one location in $H_{iS}H_{iU}$.

Having described the overall structure of the type system, we now highlight some of the more interesting rules.

The kinding rules for S(i) and *loc* encode the essence of thread-locality. The kinding rule for pair types can require the same sharability for both components without loss of expressiveness because of subkinding. We do not allow quantified types to have kinds of the form $\texttt{L}\sigma$ or $\texttt{B}\sigma$ because it is not useful to do so.

Function types are always sharable. This decision requires us to forbid functions to refer to unsharable free variables (see SR5.13). In actual Cyclone, this restriction is rather simple because free variables can refer only to functions (which are immutable) or global variables. A simpler restriction in the formalism would be to require that functions not have free variables, but then it would be impossible to use mutable references to encode recursive functions.

The definition of $\vdash_{\text{loc}}$ is a bit unsettling because it uses the absence of a kinding derivation. A more rigorous approach would be to include a "definitely unsharable" sharability, adjust the kinding rules accordingly, and use this sharability for $\vdash_{\text{loc}}$.

The rules for effect and constraint containment are exactly like in Chapter 4 except we have $\gamma; \epsilon \vdash_{\text{acc}} loc$.

Turning to the typing rules, SS5.8 and SS5.9 amplify the current capability as expected. For the former, we use locks(ℓ) because effects do not include loc and e might have the type lock(loc). In rule SS5.10, the spawned function must have the effect $\emptyset$ because threads begin holding no locks. The function's argument must be sharable because the spawned and spawning threads can access it. Unlike actual Cyclone, we allow any sharable type for the argument. Rule SR5.13 is complicated because we use $\vdash_{\text{shr}}$ to allow free references only to sharable locations. Rule SR5.16 is simple because we use $\vdash_{\text{j}}$, not the typing judgments, to restrict where junk_v can occur. Rules SR5.17 and SR5.18 give the types we would expect. In particular, newlock() has the same type as the existential package to which it evaluates.

Turning to Figure 5.14, we use $\vdash_{\text{js}}$ and $\vdash_{\text{je}}$ to establish that a term is junk-free except its next evaluation step will rewrite $x{=}\mathsf{junk}_v$ to v. As such, the only noninductive rule is for $e = x{=}\mathsf{junk}_v$. Also note that $x, v \not\vdash_{\text{je}} v'$ for any v'.

5.6.4 Type Safety

Appendix C proves this result:

Definition 5.1. *A program $P = (H; L; L_0; T_1 \cdots T_n)$ is badly stuck if it has a badly stuck thread. A badly stuck thread is a thread (L', s) in P such that there is no v such that $s = \mathsf{return}\ v$ and $L = \cdot$; and there is no i such that $H; (L; L_0, i; L'); s \xrightarrow{\text{s}} H'; \overline{L}'; s_{\text{opt}}; s'$ for some H', $\overline{L}'$, s_{opt}, and s'.*

Theorem 5.2 (Type Safety). *If $\cdot; \cdot; \cdot; \cdot; \emptyset; \tau \vdash_{\text{styp}} s$, $\vdash_{\text{ret}} s$, s is junk-free, s has no release statements, and $\cdot; (\cdot; \cdot; \cdot); (\cdot; s) \rightarrow^* P$ (where $\rightarrow^*$ is the reflexive transitive closure of $\rightarrow$), then P is not badly stuck.*

This theorem is in ways stronger and in ways weaker than theorems in earlier chapters. It is stronger because it establishes that each thread is sound, not just that *some* thread is not badly stuck. It is weaker because the type system allows deadlock. A thread can be stuck because a lock i is unavailable. In fact, the entire machine can be stuck if all threads are waiting for locks. So by definition, a thread is not badly stuck so long as it could take a step if one additional lock were available. (The definition includes threads that do not need an unavailable lock.)

5.7 Related Work

Synchronization idioms, techniques for detecting synchronization errors, and language support for multithreading are far too numerous to fully describe. This

section focuses only on the most closely related work and work that could improve Cyclone multithreading.

The Cyclone system for preventing data races is most similar to a line of work that Flanagan and Abadi began [73]. Their seminal publication used singleton lock types, lock-type annotations on mutable references, and explicit effects (they called them permissions) on function types to prevent data races for a small formal language. Their term-level constructs correspond closely to `spawn`, `sync`, and `newlock`. They present a semantics and prove that programs do not have data races, but a race could not actually make their machine stuck. They allow universal and existential quantification over lock types, but not over ordinary types. They extend the type system with a partial order that prevents deadlock.

In adapting their work to object-oriented languages [72], Flanagan and Abadi chose to use term-level variables for lock names instead of type-level variables. This change introduces a very limited form of dependent type because types now mention terms. Avoiding type variables may be more palatable to programmers, but it introduces complications. First, the variables should not stand for mutable locations, else the language is unsound because x does not always contain the same lock. (If the type of x restricts its type sufficiently, e.g., by saying it has type `lock_t<`y`>`, the result is sound, but then mutation is useless.) Second, the rules for type equivalence must use some notion of term equivalence. Determining if two terms evaluate to the same lock is trivially undecidable, so more restrictive rules are necessary. Because the programmer cannot control these restrictions, Cyclone's approach is more flexible by letting programmers gives compile-time names to locks, independent of where the locks are stored or how they are accessed.

Using term variables also affords Flanagan and Abadi some advantages. First, as in Java, all objects are (also) locks, so reusing term variables as lock names is economical. Second, using the "self variable" (`this` in Java) as a lock name better integrates their system with common notions of object subtyping. Term equality takes self variables into account. For example, if a method result is locked by the method's self variable, this variable is not in scope at the call site, but it is equivalent to say the result is locked by the variable in scope at the call site that names the object whose method is invoked.

Flanagan and Freund then adapted these ideas to Java, implemented the result, and found a number of previously unknown synchronization errors [74]. The Java system provides type constructors (classes parameterized by locks) and some support for thread-local data. Lock names are `final` (immutable) term variables. To support thread-local data, a class declaration can indicate that instances of the class cannot be thread-shared. All other classes are thread-shared; such classes cannot have mutable fields that are not guarded by locks, nor can they have fields holding thread-local objects. A thread-local class can have a thread-shared super-

class, but downcasts from a thread-shared type to a thread-local type (including the thread-local class overriding methods declared in the thread-shared class) are forbidden. Otherwise, it is not clear how the type system would enforce that data is thread-local. The focus of the work was minimizing explicit annotations and finding potential data races; it does not appear that race-freedom proofs exist.

Boyapati and Rinard developed a similar system that allows more code reuse by, in Cyclone terms, allowing *loc* to instantiate a lock-name parameter [31]. Hence whether an object is thread-local or thread-shared can depend on its class and the instantiation of the lock-name parameters of the class. The result allows just about as much code reuse as Cyclone, but they do not have an analogue of `nonlock`. The system also supports extensions described in Section 5.5, including object migration (by using unique pointers, i.e., pointers to data that cannot be aliased) and unsynchronized sharing of immutable data. However, it does not (safely) support downcasts when the target type has more lock-name parameters than the source type. In general, per-object run-time type information is necessary to check that the target type instantiates its class correctly.

Subsequent work by Boyapati, Lee, and Rinard extends the system with deadlock prevention [29]. This work also resorts to (implicit) run-time type passing as necessary to support safe downcasts. An associated report [30] explains how to avoid run-time type passing in the most common cases and how to implement the scheme on an unmodified Java Virtual Machine. Boyapati et al.'s systems do not have accompanying formalisms with type-safety proofs.

Guava [15] is another Java dialect with static data-race prevention. The class hierarchy makes a rigid distinction between thread-local and sharable objects. The latter allows only synchronized access to methods and fields. A "move operator" soundly allows object migration.

There are many other race-detection systems, some of which are dynamic [44, 46, 182, 204]. As usual, dynamic and static approaches are complementary with different expressiveness, performance, and convenience trade-offs. Because Cyclone's type safety needs data-race prevention, a static approach feels more appropriate. It is also easier to implement because there is no change to code generation.

The Cyclone system does not prove that programs are deterministic. For domains such as parallel numerical computations, such stronger guarantees help detect errors. In open systems like operating systems and servers, determinism is impossible. Nonetheless, preventing data races (on individual memory locations or objects) is insufficient for preventing application-level races. That is, an application may intend to keep several objects synchronized. If procedural abstraction controls access to such objects, then it suffices for the relevant procedures to appear *atomic*. Flanagan and Qadeer develop a type system for enforcing atomicity [77].

They also note that if the underlying memory model ensures atomic access of words, then some functions are atomic even without explicit synchronization.

Static analyses that find thread-local data can eliminate unnecessary locking in Java [4, 23, 45]. Adapting such interprocedural escape analyses to Cyclone would reduce annotations but complicate the language definition.

Other work on safe languages for low-level applications, described in more detail in Chapter 8, has not allowed threads. In Vault [55, 66], a type system that restricts aliases can track stateful properties about data at compile time. Mechanisms termed *adoption* and *focus* allow tracking state within a lexical scope without knowing all aliases of the data. This scoping technique relies crucially on the absence of concurrent access.

In CCured [164], unmodified legacy C applications are compiled unconventionally to detect all memory-safety violations. The key to good performance is a whole-program static analysis to eliminate many unnecessary run-time checks. The analysis assumes the program is single-threaded. With arbitrary thread interleavings, we would expect much more conservative results. Moreover, the run-time checks themselves are not thread-safe. Making them so would require expensive synchronization or precise control of thread scheduling.

The Warlock system [191] is an older, unsound approach to static race detection for C programs. Two factors violate soundness. First, it analyzes C programs and simply assumes they are memory safe. Second, it uses mutable variables for lock names. Hence it will wrongly conclude that a program is race-free if two threads acquire the lock at x before reading y even though the lock at x has been changed in-between the two acquisitions. For a bug-finding tool, this unsoundness may be bearable because nonmalicious programs might rarely have this sort of mistake.

The only work I am aware that combines multithreading with safe memory management is the Real-Time Specification for Java [20]. As described in Chapter 4, this Java extension has regions with lexically scoped lifetimes and attempting to create a reference from an older region to a younger one is a run-time error. As in Cyclone, one thread's oldest region can appear in another thread's stack of regions. The region is not deallocated until every thread is done with it. In other words, Cyclone and Real-Time Java support thread-shared regions the same way. Because the Real-Time Java type system has no notion of lifetime, thread-shared regions cause no complications whereas in Cyclone they lead to subtyping restrictions.

Chapter 6

Uninitialized Memory and NULL Pointers

This chapter describes how Cyclone prevents reading uninitialized memory and dereferencing `NULL` pointers.

Allowing such operations can easily compromise safety. When allocating memory for a pointer, (e.g., `int* x; *x=123;`), C and Cyclone do not specify an initial value for `x`. In practice, implementations often leave the value that the memory contained when it was used previously, possibly an arbitrary `int`. When dereferencing a `NULL` pointer, C has unspecified behavior. Most implementations implement `NULL` as 0, so if `x` is `NULL` and has type `struct T*`, we expect `x->f=123` to write to an address corresponding to the size of the fields preceding `f`. (Because this size may be large, it may not suffice to make low addresses inaccessible.) We could insert a check for `NULL` and raise an exception, but performance and reliability encourage the elimination of redundant checks. For simplicity, this chapter usually assumes the implementation does not insert implicit checks, so programs that might dereference `NULL` pointers are rejected at compile-time.

To solve these two problems, we use techniques that differ substantially from those used to solve problems in earlier chapters. For types, regions, and locks, our solutions relied on *invariance*: Throughout an object's lifetime, we require it has the same type, region, and lock. Although some safe programs do not maintain such invariants, these restrictions seem reasonable and help make undecidable problems tractable.

For the problems in this chapter, invariance is too restrictive. It amounts to requiring immediate initialization (e.g., forbidding declarations that omit initializers), which can hurt performance and makes porting C code more difficult. For pointers, it *is* often sensible to enforce a "not-`NULL`" invariant and Cyclone provides this option. However, many idioms, such as linked lists, use `NULL`. Given a

possibly-`NULL` pointer, we must allow programs to test at run-time whether it is actually `NULL` and, if not, dereference it. Hence, both problems warrant solutions that determine program-point specific (i.e., *flow-sensitive*) information. A variable that is possibly uninitialized at one point can be initialized after an assignment. A variable that is possibly `NULL` at one point can be assumed not-`NULL` after an appropriate test, subject to caveats due to aliasing.

Therefore, this chapter develops a sound intraprocedural flow analysis. Because flow analysis is a mainstay of modern language implementations, Section 6.1 describes the novel issues that arise with Cyclone, particularly under-specified evaluation order and pointers to uninitialized data. Section 6.2 presents the analysis informally, focusing on how it interprets code as transforming an abstract state. Section 6.3 evaluates the approach and describes two sophisticated examples. Section 6.4 defines a formal abstract machine and a declarative type-theoretic formulation of the flow analysis. This precision is valuable given the sophistication of the analysis, but the connection between the declarative formulation and the analysis algorithm remains informal. Because of the machine's dynamic semantics, soundness (proven in Appendix D) implies well-typed programs cannot attempt to dereference `NULL` pointers or destruct "junk" values that result from reading uninitialized memory. Section 6.5 discusses related work on source-level flow analysis.

6.1 Background and Contributions

A simple dataflow analysis that approximates whether local variables are initialized or (not) `NULL` is a straightforward application of well-known techniques. However, several important issues complicate the analysis in Cyclone:

- The analysis is for a general-purpose source language and is part of the language's definition, so it is inappropriate to define the analysis in terms of a simplified intermediate representation.
- The analysis is for a language with under-specified evaluation order.
- The analysis reasons about pointers to particular locations, including uninitialized ones.
- The analysis is for a language with exceptions and exception handlers, which increases the number of possible control transfers.
- The analysis reasons about `struct` fields separately. Doing so significantly complicates the implementation, but turns out to be an orthogonal and less interesting issue.

Section 6.1.1 describes a simple flow analysis as background and to introduce some Cyclone terminology. It is purposely unoriginal. Sections 6.1.2 and 6.1.3 describe the problems surrounding pointers and evaluation order, respectively. The solutions in Section 6.2 are the important technical aspects of this work.

6.1.1 A Basic Analysis

An intraprocedural dataflow analysis can assign each local variable an abstract value (which we will call an *abstract rvalue* for reasons explained in Section 6.2) at each program point in a function body. For `NULL` pointers and initialization, this domain of abstract rvalues makes sense:

- NONE: A possibly uninitialized variable
- ALL$*$: An initialized variable that may be `NULL`
- ALL@: An initialized variable that is definitely not `NULL`
- 0: An initialized variable that is definitely `NULL`

This domain forms a lattice where $r_1 \leq r_2$ means r_2 is more approximate than r_1. This partial order is the reflexive, transitive closure of the relation holding $0 \leq$ ALL$*$, ALL@ $\leq$ ALL$*$, and ALL$* \leq$ NONE.

A map from variables to abstract rvalues is an *abstract state*. We can interpret statements as transforming abstract states. For example, if the abstract state at the program point before the assignment `x=y` maps `y` to ALL$*$ and `x` to NONE, then the abstract state after the assignment is like the one before except `x` maps to ALL$*$. Declaring a variable `z` extends the abstract state by mapping `z` to NONE. Statements can make the analysis *fail*, such as `*x` if `x` does not map to ALL@ or `f(x)` if `x` maps to NONE. Finally, tests can *refine* an abstract state. For example, if `x` maps to ALL$*$ before `if(x)` s_1 `else` s_2, we can sometimes map `x` to ALL@ before s_1 and to 0 before s_2.

A program point's abstract state should approximate the abstract state of all its control-flow predecessors. For example, if after one branch of a conditional statement `x` maps to ALL@ and after the other branch `x` maps to ALL$*$, then the abstract state for after the conditional should map `x` to ALL$*$.

Control-flow cycles (e.g., loops) require the analysis to iterate because we cannot always compute the abstract states for a program point's control-flow predecessors prior to analyzing the statement at the program point. For example, for `while(`e`)`s, if we have abstract state Γ before the loop and Γ' after the loop body s, we must reanalyze the loop with an abstract state before the loop approximating

Γ and Γ'. If Γ is such an abstract state, we are done. The analysis always terminates because there is no infinite sequence of abstract states where each element is strictly more approximate than the previous one.

By giving an appropriate abstract meaning to each statement as just sketched, if the analysis does not fail, then we know executing the function cannot dereference a `NULL` pointer or use uninitialized memory.

6.1.2 Reasoning About Pointers

The description above ignored the analysis of code that creates, initializes, and uses non-`NULL` pointers. If `x` maps to ALL$*$, we can say `y=&x` transforms the abstract state so `y` maps to ALL@, but this ignores the fact that we create an alias for `x` (namely `*y`). For example, this code is not safe:

```
void f(int *x) {
  int **y;
  y = &x;
  if(x) {
    *y = NULL;
    *x = 123;
  }
}
```

One solution involves making worst-case assumptions for variables after their address is taken. Indeed, our analysis will enrich abstract states with *escapedness* information to track if unknown aliases to a memory location exist. However, in the example above, this conservatism is unnecessary because at each program point our analysis can determine exactly the aliases of `x`.

In fact, tracking aliases appears crucial for supporting `malloc` in a principled way. Consider this simple safe example:

```
void f(int *p) {
  int ** x;
  x = malloc(sizeof(int*));
  // **x = 123;
  *x = p;
  **x = 123;
}
```

The assignment to `x` makes `x` point to uninitialized memory. So accessing `**x` is unsafe before the assignment to `*x`. The abstract rvalues presented so far are

ill-equipped to analyze code using `malloc`. After the assignment to `x`, the only safe choice would be NONE, but this choice renders `malloc` useless, rejecting the ensuing assignment to `*x`.

Our solution adds abstract rvalues describing points to named abstract locations. This solution significantly complicates the notion of one abstract state approximating another. However, it leads to a more powerful system than conventional flow analyses for preventing safety violations.

Finally, an intraprocedural analysis with limited alias information is ill-suited to track properties of large data structures such as lists. For safety, it suffices to require all such data to be initialized and have abstract rvalue ALL$*$. However, if a data structure has an invariant that some data is never `NULL`, this solution cannot check or exploit this invariant. Therefore, we enrich the *type system* with types of the form τ@ to describe not-`NULL` pointers. Section 6.2 explains how the types τ@ and $\tau*$ interact with the abstract rvalues ALL@ and ALL$*$.

In summary, our analysis adds escapedness information, points-to information, and interaction with not-`NULL` types. Together, these additions add significant expressiveness. For example, Section 6.3 presents code to create circular doubly-linked lists using this type:

```
struct CLst {
  int val;
  struct CLst @ prev;
  struct CLst @ next;
};
```

Most safe languages do not have a way to create circular data structures with such invariants.

6.1.3 Evaluation Order

Cyclone and C do not fully specify the order of evaluation of expressions: Given `f(`e_1`,`e_2`)`, we cannot assume e_1 evaluates before e_2. This flexibility complicates defining a sound flow analysis. For example, we must reject `if(x) f(*x,x=NULL)`. This section defines several variations of the problem because different solutions in the next section work for different variations. For example, only some of the variations consider `f(x && *x, x=NULL)` a safe expression.

As the most lenient, the "actual C semantics" does not require any order between so-called *sequence points*. For example, given `e1(e2(e3),e4)`, there are more than 24 (i.e., 4!) legitimate evaluation orders. (Even after evaluating `e2` and `e3`, we can do the inner function call at various times with respect to `e1` and `e4`.) Certain expressions do restrict evaluation order. For example, for `e1,e2`,

the comma operator ensures all of `e1` is executed before any of `e2`. Given `e1 + (e2,e3)`, there remain at least 3 legitimate evaluation orders. To make matters worse, C forbids expressions that are nondeterministic because of evaluation order. Specifically, if a read and a write (or two writes) to the same location are not separated by a sequence point, then the program is illegal and the result is implementation-dependent (unless the read is to determine the value for the write). That is, in such cases, a standards-compliant implementation can perform arbitrary computation.

A safer alternative is "C ordering semantics." We allow all the evaluation orders as C, but we do not deem reads and writes (or two writes) to the same location between sequence points illegal. Put another way, an implementation cannot use the "actual C semantics" to assume (lack of) aliasing that may not hold. It must execute all expressions correctly, but the order of evaluation remains very lenient. This alternative is what Section 6.4 formalizes, but the formalism has no sequence points within expressions.

A less lenient alternative is "permutation semantics." Given a function call `e1(e2,...,en)`, an implementation can execute the n expressions in any order, but it cannot execute part of one and then part of another. Similarly, assignment statements and operators like addition would allow left-then-right or right-then-left. Although this semantics is not C, it is the rule for languages such as Scheme [179] and OCaml [40], so it is a problem worth investigating.

We could eliminate the issue entirely with a "deterministic semantics." Like Java [92], we could define the language such that expressions like function calls evaluate subexpressions left-to-right (or another fixed order).

Another less lenient approach is to enforce a "purity semantics" by forbidding expressions that write to memory. Making assignments statements instead of expressions is insufficient because a function call could include assignment statements (and could mutate the caller's local variables if passed pointers to them).

In general, less lenient approaches transfer the obligation of proving optimizations safe from the programmer to the implementation. As examples, fixing evaluation order can increase register pressure, and allowing writes to aliased locations can restrict instruction selection. Because conventional compilers perform such optimizations after the compiler has chosen an evaluation order, these issues are cleanly separated. In the actual Cyclone implementation, the compiler produces C code and then invokes a C compiler. As such, Cyclone must preserve safety even though its target language does not have a fixed evaluation order. Although technically incorrect, the implementation assumes "C ordering semantics."

6.2 The Analysis

This section presents the essential aspects of Cyclone's flow analysis. We begin with a description of abstract states, explaining how they capture escapedness, must points-to information, and not-`NULL` invariants. We then explain the analysis of expressions, where the key ideas are the use of not-`NULL` types and the evaluation-order problems. Next we explain the analysis of statements, focusing on how to join two abstract states and how tests can refine information. We delay the description of some relevant language features (aggregates, recursive types, `goto`, and exceptions) until the end.

6.2.1 Abstract States

An abstract state maps each *abstract location* to a type, an *escapedness*, and an *abstract rvalue*. Abstract locations are allocation sites, which include (local) variables and occurrences of `malloc`. (Because dangling pointers are not our present concern, we can think of `malloc` as declaring a local variable and evaluating to its address. We just "make up a distinct variable" for each `malloc` occurrence.)

For now we consider only types of the form int, $\tau*$ and $\tau@$ where τ is a type and $\tau@$ cannot describe `NULL`. An escapedness is either ESC (escaped) or UNESC (unescaped); the former means the aliases for the location may not be known exactly. For example, the abstract state for the program point after `if(flip()) x=&y; else x=NULL;` must consider `y` escaped. Abstract rvalues are either NONE, ALL@, ALL$*$, 0, or $\&x$ where x is an abstract location. We explained all but the last form previously. The abstract rvalue $\&x$ describes only values that *must point to* the location (most recently) produced by the allocation site x. Hence, pointer information is an inherent part of our abstract domain. When a location is allocated, its escapedness is UNESC, its abstract rvalue is NONE, and its type is provided by the programmer (even for `malloc`).

If a location is escaped, it becomes difficult for the analysis to track its contents soundly because it is not known which assignment statements mutate it. Therefore, an abstract state is ill-formed if an escaped location does not have the abstract rvalue appropriate for escaped locations of its type. In particular, it must have ALL$*$ unless the type is $\tau@$, in which case it must have ALL@. The analysis fails if no well-formed abstract state describes a program point. For example, it rejects:

```
void f(int x) {
  int *p1, **p2;
  if(x)
    p2 = &p1;
}
```

This function is safe, but at the end, `p1` is escaped but uninitialized, so no well-formed abstract state suffices.[1]

Given abstract states Γ and Γ', we need an appropriate definition of Γ' being *more abstract* than Γ, written $\Gamma \leq \Gamma'$. They must have the same domains and map each abstract location to the same type because the allocation sites and types are invariant. For each x, we require Γ' map x to a more approximate escapedness (UNESC $\leq$ ESC) and a more approximate abstract rvalue. In addition to the approximations in Section 6.1, we can add $\&x \leq$ ALL@, but only if Γ' considers x escaped. After all, this approximation "forgets" an alias of x. As defined, $\Gamma \leq \Gamma'$ does not imply Γ' is well-formed, even if Γ is well-formed. The section on statements describes a *join* operation that either fails or produces a well-formed approximation of two well-formed abstract states.

6.2.2 Expressions

As in previous chapters, we analyze left-expressions and right-expressions differently. In each case, given an abstract state and an expression, we produce an abstract state describing how the expression transforms the input state (due to effects like assignments). For right-expressions, we also produce an abstract rvalue describing the expression's result. For left-expressions, we produce an *abstract lvalue*, which is either some location `x` or ?, representing an unknown location. We describe some of the more interesting cases before discussing evaluation-order complications.

The only left-expressions we consider here have the form `x` or `*`e where e is a right-expression. The former produces the abstract lvalue `x` and does not change the abstract state. For the latter, we analyze e to produce an abstract rvalue r and an abstract state that is our result. If r is ALL@, then our abstract lvalue is ?—we do not know the location. If r is $\&x$ for some x, then the abstract lvalue is x. Else we fail because we might dereference `NULL` or an uninitialized pointer.

The analysis of many right-expressions is similar. For example, `NULL` abstractly evaluates to 0 and does not transform the abstract state. For a variable x, we look up its abstract rvalue in the abstract state. The resulting abstract rvalue for `&`e is either $\&x$ for some x or ALL@, depending on the analysis of the left-expression e. A function call fails if an argument has abstract rvalue NONE.

The analysis of the right-expression $*e$ is more interesting because the resulting abstract rvalue might depend on the type of e: If e has abstract rvalue $\&x$, then we look up the abstract rvalue of x in the context. Else if $*e$ has some type τ@,

[1] Giving `p1` and `p2` abstract rvalue NONE (instead of giving `p2` abstract rvalue `&p1`) suffices, but our analysis fails instead.

we can conclude ALL@ even if we all we know about e is that it is initialized. If e is uninitialized, we fail. Else the result is ALL$*$.

The other interesting case is assignment e_1=e_2. If e_1 abstractly evaluates to a location x that is unescaped, then we can change the abstract rvalue for x to the resulting abstract rvalue for e_2. If the abstract lvalue is ? or an escaped x, then the abstract rvalue for e_2 must be the "correct" one for the type of e_1 (either ALL$*$ or ALL@). Note that this rule still lets an escaped e_1 have type τ@ and e_2 have type $\tau*$ if e_2 abstractly evaluates to ALL@.

Our descriptions of function calls and assignments have ignored their under-specified evaluation order. For example, it is unsound to analyze `f(*x,x=NULL)` assuming left-to-right evaluation because the abstract state used to analyze `*x` is too permissive. We discuss several alternatives and determine their soundness under the various semantics defined in Section 6.1.

Determinization: It is easy to translate Cyclone to C (or C to C) in a way that gives a "deterministic semantics" (e.g., left-to-right). However, the translation must introduce many local variables to hold temporary results. For example, `f() + g();` would become something like `int x=f(); int y=g(); x+y;`. In this particular example, the original expression is safe, even under "actual C semantics," so it is necessary to introduce local variables only if we define Cyclone to have a deterministic semantics. The obvious advantage of determinizing the code is that it makes any problems with under-specified evaluation order irrelevant. However, many languages continue to have more lenient semantics, so rather than resort to determinization, we investigate other options. Also, when using a target language such as C, maintaining a "deterministic semantics" can lead to longer compile times and slower generated code. These effects may be tolerable in practice.

Exhaustive Enumeration: A simple way to analyze an expression soundly is to analyze it under every possible evaluation order, ensure it is safe under all of them, and take the join over each resulting typing context (and abstract rvalue) to produce the overall result. However, this approach can be computationally intractable. For example, under "permutation semantics," analyzing a function call with n arguments requires $n!$ permutations. Furthermore, some of the n arguments may themselves suffer from a combinatorial explosion that would manifest itself in each of the outer context's $n!$ checks. Under "C ordering semantics" even more possibilities exist. The approach is insufficient for "actual C semantics" because ill-defined C programs have an infinite number of possible evaluations.

Perhaps a compiler could use heuristics to achieve the precision of the combinatorial approach in practice without suffering intolerable compile times. For

example, many expressions with huge numbers of potential evaluation orders are probably pure in the sense of the next approach.

Purity: If an expression does not write to memory, then evaluation order affects neither safety nor an expression's result. Ignoring function calls, prohibiting writes within expressions with under-specified evaluation order is probably reasonable. However, it does prevent common idioms including `x=y=z` and `a[i++]`. Except under "actual C semantics," we do not need to prohibit all writes. It suffices to prohibit writes that change the abstract state.

Prohibiting even these writes is more restrictive than requiring the analysis of an expression like a function argument to conclude the same abstract state with which it started. We explain why with the next approach. Furthermore, it is useful to allow expressions to change the typing context if doing so does not make other expressions (that may execute before or after) unsafe. Our final approach ("changed sets") addresses this issue.

Taking Joins: Given an expression like `f(`e_1`,`e_2`,...,`e_n`)`, suppose we analyzed e_1 under an abstract state Γ to produce Γ_1, e_2 under Γ to produce Γ_2, and so on. We could then use the join operation described with the analysis of statements to produce an abstract state Γ' that is more abstract than Γ and each Γ_i. If Γ' were strictly more approximate than Γ, we could iterate with Γ' in place of Γ, else we could use Γ' (i.e., Γ) as the resulting abstract state.

Because we keep iterating with more approximate abstract states until no expression causes a change, this procedure essentially analyzes the n expressions as though they might each execute any number of times. This interpretation is more approximate than exploiting that, in fact, each expression executes exactly once, so it is clearly sound under a "permutation semantics." Less obviously, it is sound under a "C ordering semantics" if and only if we do not have expressions with sequence points (namely C's `&&`, `||`, `?:`, and comma operators). In other words, it is not sound.

For example, consider the expression `f(x=NULL,(x=&y,*x))`. With the join approach, we can conclude an abstract state where `x` has abstract rvalue ALL$*$. But with "C ordering semantics," the code is unsafe because it might assign `NULL` to `x` just before dereferencing it. Under "permutation semantics," the code is safe.

To restore soundness, we have several options. First, we could use the purity approach instead. In other words, we would type-check expressions with sequence points more strictly when they appeared in positions where they may not execute without interruption. Second, we can try to allow a sequence expression to change the flow information if no other expression might invalidate the change. This

approach would reject `f(x=NULL,x=&y,*x)`, but would allow `f(37,(x=&y,*x))`. The next approach describes how we might do so soundly.

Changed Sets: Under the join approach (and the purity approach), there is no way for the subexpression of an expression with under-specified evaluation order to affect the typing context after the expression. Hence we do not consider `int x; f(37,x=3);` to initialize `x` because the abstract rvalue for `x` must remain NONE.

In actual Cyclone, one common result of this shortcoming is unnecessary implicit checks for `NULL`: With a command-line option, programmers can permit dereferencing possibly-`NULL` pointers. In this case, the implementation may insert an implicit check for `NULL` and potentially raise an exception. However, we can still use the flow analysis to avoid inserting checks where necessary. For example, given `*x=1;*x=2`, the second assignment does not need a check (assuming `x` has escapedness UNESC) because if `x` is `NULL`, the first assignment would have thrown an exception. In practice, code of the form `f(x->a); g(x->b);` is quite common. To use the flow analysis to avoid checking for `NULL` before `x->b`, we must use the earlier check `x->a`, but it appears in an under-specified evaluation-order position.

A small enhancement achieves the necessary expressiveness: We iterate as in the "taking joins" approach, but we also maintain a "changed set" for each expression. This set has the unescaped locations for which the expression changes their abstract rvalue. If a location appears in only one changed set, it is safe to use the abstract rvalue it had after the corresponding expression executed. (We can use the change even if other expressions changed the location provided that they all changed the expression to the same abstract rvalue.)

Compared to the join approach (or the purity approach), this enhancement acknowledges that all the expressions do execute ("at least once" if you will), so we can incorporate their effects in the resulting abstract state. Because we use such changes only in the final result—not in the abstract state with which we iterate—we still reject expressions like `f(*x,x=NULL)`.

6.2.3 Statements

The analysis takes a statement and an abstract state approximating the state after all control-flow predecessors and produces a sound abstract state for the point after the statement. This procedure is completely conventional: The result for $s_1; s_2$ is the result for s_2 given the result for s_1, the result for `if(`e`)`s_1 `else` s_2 is the join of the results for s_1 and s_2, and so on. (For a less dismissive description, see Section 6.4.) The interesting subroutines are the analysis of test expressions and the computing of joins.

For test expressions (as in conditionals and loop guards), one safe approach is to analyze the expression as described in the previous section and use the abstract state produced as a result for both control-flow successors. However, this approach does not refine the abstract state, so we endeavor to be less conservative where possible. Specifically, if the text expression is `x` (e.g., `while(x)` s) and `x` is unescaped and has abstract rvalue ALL$*$, then we can analyze the "true" successor (s in our example) with `x` having abstract rvalue ALL@ and the "false" successor (after the loop in our example) with `x` having abstract rvalue ALL$*$.

We do not require that the test is exactly `x` to refine the abstract states for successors. After all, we would like an analysis that is robust to certain syntactic equalities, such as writing `x!=NULL` or `NULL==x` instead of `x`. Fortunately, the abstract rvalues and abstract lvalues produced by the analysis for right-expressions and left-expressions provide exactly what we need: Given a test of the form e_1`==`e_2 or e_1`!=`e_2, if the abstract rvalue for e_1 or e_2 is 0, then we can act as though the test were syntactically simpler (after accounting for any effects we are simplifying away). Next, if a test has the form e or `!`e and the analysis for left-expressions could give e the abstract lvalue `x`, then we can treat the test as being just `x` or `!x` (again after accounting for effects). Together, these techniques provide a reasonable level of support for tests.

More pathologically, if the entire test has abstract rvalue 0 or ALL@, then the analysis determines the control flow at compile-time and can choose any abstract state for the impossible branch. (As usual with iterative analysis, we propagate an explicit $\bot$ that every abstract state approximates.) Although this addition has dubious value in source programs, it simplifies the iterative analysis, it is natural, and it is analogous to similar support in Java (as discussed in Section 6.5).

We now turn to computing a (well-formed) join for two (well-formed) abstract states, which we must do when a program point has multiple control-flow predecessors. The key issue is that locations may escape as a result of a join. For example, if `x` and `y` have escapedness UNESC and abstract rvalue ALL$*$ before `if(flip()) x=&y;`, then `y` is escaped after the conditional (and `x` has abstract rvalue ALL@). Furthermore, if `y` had had abstract rvalue `&z`, then `z` must also be escaped afterward. We now describe the algorithm that ensures these properties.

The two input abstract states should have the same domain and the same type for each location. For each `x`, we compute a "preliminary" escapedness k and abstract rvalue r as follows: Let k_1 and k_2 be the escapednesses and r_1 and r_2 be the abstract rvalues for `x` in the inputs. Then the preliminary k is ESC if and only if one of k_1 and k_2 is ESC. As for r:

- If r_1 or r_2 is NONE, then r is NONE.
- Else if r_1 or r_2 is ALL$*$, then r is ALL$*$.

- Else if exactly one of r_1 and r_2 is 0, then r is ALL$*$.
- Else if r_1 and r_2 are the same, then r is r_1.
- Else r is ALL@.

Furthermore, if r_1 or r_2 is some `&y` and r is not `&y`, then we put `y` in an "escaped set" containing all such locations we encounter while producing the preliminary k and r for each `x`. We then use the escaped set to modify our preliminary abstract state: While the set is not empty, remove some `y`. If `y` is unescaped, change it to escaped. If its abstract rvalue is NONE, *fail* because we cannot produce a sound well-formed result. Else change the abstract rvalue for `y` to the correct one for escaped locations of its type (either ALL$*$ or ALL@). If the old abstract rvalue was `&z`, add `z` to the escaped set.

This process terminates because each time we remove an element of the "escaped set" we either change a location's escapedness from UNESC to ESC or we reduce the set's size. The result becomes only more approximate at each step. The point of the "escaped set" is to make escaped exactly what we need to such that the result is well-formed. Finally, note that the procedure can soundly subsume a cycle of known pointers to a collection of unknown but initialized pointers.

6.2.4 Extensions

Having described the interesting features of the analysis, we now consider complications (and lack thereof) encountered when extending the analysis to the full Cyclone language.

Aggregate Values: Given an allocation site for a `struct` type, we track each field separately. (If a field is itself a `struct` type, then we track its fields too, and so on inductively.) To do so, we enrich our abstract rvalues with abstract aggregates. For example, if `x` has type `struct T{int* f1; int*f2;};`, then allocating `x` maps `x` to the abstract rvalue NONE $\times$ NONE. If the analysis of e produces $r_1 \times r_2$, then the analysis of e`.f1` produces r_1.

Similarly, the escapedness information for `x` would have the form $k_1 \times k_2$. If an alias to the aggregate `x` escaped, the escapedness would be ESC $\times$ ESC. But if only one field escaped (e.g., due to `&x.f1`), an abstract state could reflect it. The notions of $k_1 \leq k_2$ and $r_1 \leq r_2$ extend point-wise (covariantly) through aggregates.

Abstract lvalues can take the form `x.f1.f2...fn` (and abstract rvalues the from `&x.f1.f2...fn`). Because of aggregate assignment, we still allow `x` even if `x` has a struct type. So an assignment to (part of) a known, unescaped location can change all or part of the location's abstract rvalue. The appropriate abstract

rvalue for an escaped location with a `struct` type is the natural extension of the rules for τ@ and $\tau*$ to aggregates.

Recursive Types: Somewhat surprisingly, recursive types require no change to the analysis and cannot cause it to run forever. Essentially, the "depth" of points-to information in an abstract state is bound by the (finite) number of allocation sites in a function. Creating a data structure of potentially unbounded size requires a loop (or recursion), i.e., the reuse of an allocation site.

A subtle property of the analysis is that an abstract state always describes the most recent location that an allocation site has produced. (We can prove this property by induction on how long the iterative analysis runs. The intuition is that at the program point before the allocation site for `x`, it is impossible for the abstract state to indicate that some `y` has abstract rvalue `&x`.) So the analysis naturally loses the ability to track multiple locations that an allocation site creates. Section 6.3.5 describes how the analysis works for a loop that creates a list.

The analysis can even track cycles, which are impossible without recursive types. Nothing prevents the abstract state from having a cycle of must-points-to information. When a cycle escapes, the join operation ensures the entire cycle will escape. We argued above that this operation terminates.

Goto: Unstructured control flow (`goto`,`break`,`continue`) poses little problem for iterative flow analyses, including ours. If the abstract state before a jump is some Γ, then the analysis must analyze the target of the jump under an abstract state more approximate than Γ. Jumps can cause loops, so the analysis may iterate. As usual, the implementation stores an abstract state for each "jump target" and tracks whether another iteration is necessary.

Exceptions: Integrating exceptions is also straightforward, but the algorithm is conservative about when an exception might occur. Cyclone has statements of the form `try` s `catch {case` p_1`:` s_1 `...case` p_n`:` s_n`}`. Within s, the expression `throw` e transfers control to s_i provided that e evaluates to an exception that *matches* p_i and no statement within s *catches* the exception. Given that s_i executes only if an exception occurs, it seems reasonable to check it under rather conservative flow information.

Therefore, we check s_i under an abstract state that is more approximate than *every* abstract state used to type-check a statement or expression in s. (Section 6.5 explains why Java's analysis can just use the abstract state before s.) Because a function call executed within s can terminate prematurely with an exception, it is important that our analysis soundly approximates the flow information when such

exceptions are thrown, even though the analysis is intraprocedural. The key is to require a location x to have escapedness ESC if x is reachable from an argument to a function. Put another way, a function argument e is checked much like an assignment of e to an unknown location. By requiring ESC, the analysis is sound regardless of what the function call does or when it throws an exception.

Type-checking `throw` e is simple: We require that e safely evaluates to an initialized exception. It is sound to produce any abstract state (we use an explicit $\perp$).

6.3 Evaluation

Having informally defined the analysis, we now evaluate the result qualitatively. The formalism argues the analysis is sound, so here we focus on how expressive it is. We begin by admitting how the actual Cyclone implementation is more lenient (though still safe) and considering if it would not be better just to rely on run-time techniques for initialization and `NULL` pointers. We then focus on the most important idioms that the analysis permits (Section 6.3.3) and does not (Section 6.3.4). The next two sections present two more sophisticated examples. Finally, Section 6.3.7 describes an extension for supporting a simple form of interprocedural initialization.

6.3.1 Reality

The actual Cyclone implementation is more lenient than this chapter has thus far suggested. The differences are not interesting from a technical perspective, but they are more convenient for programmers without sacrificing safety.

First, we do not require initializing numeric values before using them. Using "junk bits" leads to unpredictable program behavior, but it does not violate memory safety. It does allow reading values from previous uses of the memory now used for a number, which could have security implications. The main reason for this concession is the lack of support for arrays. It allows omitting an initializer for a character buffer. For some text-manipulation programs, an extra pass through a buffer to initialize it can hurt performance.

Second, if a sequence of zeros is appropriate for a type (i.e., the type has no non-`NULL` components), then programmers can use `calloc` to initialize memory with that type. For example, `calloc(n*sizeof(int*))` creates an initialized array of length n. Replacing `int*` with `int@` is illegal.

Third, a compiler option allows dereferences of possibly `NULL` pointers. With this option, the compiler inserts implicit checks and raising an exception upon

encountering `NULL`. In terms of our formalism, we allow $*e$ even if e has abstract rvalue ALL$*$. When e is unescaped, we can use the dereference to refine the location's abstract rvalue to ALL@, just like for tests. Intuitively, if the dereference does not raise an exception, x is not `NULL`.

It is tempting to allow `&x->f` (i.e., `&((*x).f)`) without checking if `x` is `NULL` because the expression does not actually dereference `x`. (By analogy, Chapter 4 allows `x` to point to deallocated storage.) However, allowing such expressions makes it difficult if not impossible to check for `NULL` when a later dereference occurs. Because `*&x->f` is not 0, we might naively not raise an exception for `*&x->f`, even if `x` is 0. Therefore, we check for `NULL` even under the address-of operator.

6.3.2 Run-Time Solutions

Considering the complexity of the flow analysis and its limitations, it is worth asking whether Cyclone should simply initialize memory and check for `NULL` at run-time. The implementation could still optimize away initializers and checks that were provably unnecessary. While sacrificing programmer control and compile-time error detection (two primary motivations for this dissertation), we would gain simplicity.

Implicit initialization for Cyclone is more difficult than for C or Java precisely because we have not-`NULL` types. For these languages, if `NULL` is implemented as 0, then a sequence of zeros is appropriate for every type. Therefore, it is trivial to find an initializer for any type of known size.

For Cyclone, it should be possible to invent initializers, but it is not simple. Given τ@, we would need to create initialized memory of type τ and take its address. For recursive types, we must not create initializers of infinite size. For function types, we must invent code. (The function body could raise an exception for example.) For abstract types, some part of the program that knows the type's implementation must provide the initializer. This technique basically amounts to having default constructors for values of abstract types and implicitly calling the constructors at run-time.

6.3.3 Supported Idioms

Despite the complexity of the analysis, it is most often useful for simple idioms separating the allocation and initialization of memory. For example, it accepts this code, assuming s_1 and s_2 do not use `x`.

```
int *x;
if(e) { s1; x=e1; } else { s2; x=e2; }
f(x);
```

One might argue that uninitialized local variables are poor style. But requiring unused initializers just makes incorrect C programs easier to debug because run-time errors are more predictable. With a sound analysis, it is better to omit unnecessary initializers when possible because the analysis proves the initializer is useless. Hence omitting the initializer better describes what the program does and is marginally more efficient. One might argue instead that the real problem is C's distinction between expressions and statements. In many languages, we could write an initializer for `x` that did the same computation as the if-statement, but without function calls, C's expression language is too weak. Restructuring the code in this way amounts to a more functional style of programming. Given C's commitment to imperative features, expanding the language of initializers seems like more trouble than it is worth.

Our next example is a straightforward use of `malloc`:

```
struct Pr { int *x; int *y; };
struct Pr* f(int *a, int *b) {
  struct Pr* ans = malloc(sizeof(struct Pr));
  ans->x = a;
  ans->y = b;
  return ans;
}
```

We can expect code like this example whenever porting a C application that uses heap-allocated memory. It requires we track the fields separately and use must points-to information. Without resorting to *ad hoc* restrictions such as, "heap-allocated memory must be initialized immediately after it is created," the analysis naturally subsumes this common case.

Our next example uses not-`NULL` types:

```
void f(int *x, int @y, int b) {
  if(b) {
    x  = y;
    y  = x;
    *y = *x;
  }
  y  = x;  // illegal
  *y = *x; // illegal
}
```

In the body of the if-statement, we can assign `y` to `x` because `int*` is a subtype of `int@`; it is always safe to treat a value of the latter as a value of the former. Moreover, the flow information after the assignment notes that `x` contains ALL@ instead of ALL$*$. After the if-statement, these assignments may not have occurred so both assignments are illegal.

This more interesting example uses a run-time test to determine whether a pointer is `NULL`.

```
int f(int *x) {
  if(x)
    return *x;
  else
    return 42;
}
```

We can refine the abstract value of `x` after the test because function parameters are initially unescaped.

The function `f` is a suitable auxiliary function for programmers that want to dereference `int*` pointers without concern for compile-time assurances or performance. If all programmers had these desires, the compiler could simply insert implicit checks before every memory dereference. Instead, programmers can safely avoid redundant checks, as this example shows:

```
struct List<α> { α hd; struct List<α> * tl; };
int length(struct List<α> * lst) {
  int ans=0;
  for(; lst != NULL; lst = lst->tl) ++ans;
  return ans;
}
```

Before reading `lst->tl`, we need not check whether `lst` is `NULL` because on every control-flow path to the dereference, the test `lst != NULL` has succeeded.

Finally, the must points-to information allows simple copies like the following:

```
struct Pr { int *x; int *y; };
struct Pr* f(int *a, int *b) {
  struct Pr* ans = malloc(sizeof(struct Pr));
  int ** q = &ans->x;
  struct Pr* z = ans;
  *q = a;
  z->y = b;
  return ans;
}
```

The point is that must points-to information captures certain notions of aliasing. For example, because `ans` points to the allocated memory, `&ans->x` points to the memory's first field, so the initialization of `q` makes `q` point to the first field. Therefore, `*q=a` initializes the first field. Such convoluted code may not deserve specific support, but it is a by-product of a uniform set of rules not subject to syntactic peculiarities.

6.3.4 Unsupported Idioms

This section focuses on conservatism arising from aliasing, path-insensitivity, lack of interprocedural support, and (most importantly) lack of array support.

Aliasing and Path-Insensitivity: First, despite using must points-to information, the analysis still treats pointers quite conservatively. This conservatism often arises with code like the following:

```
void f(int *@x) {
  if(*x != NULL)
    **x = 123; // safe but rejected
}
```

We reject this program because `x` has abstract rvalue ALL@, so the analysis does not reason precisely about `*x`. Although this code is safe, similar examples are not because of aliasing:

```
void f(int *@x, int *@y) {
  if(*x != NULL) {
    *y = NULL;
    **x = 123; // unsafe if x==y
  }
}
```

Even an intervening function call is problematic because `*x` might refer to a global variable.

It is also possible not to know all aliases of a local variable:

```
void f(int b) {
  int *x;
  int **y;
  if(b)
    y = &x;
  s // rejected because uninitialized memory escapes
}
```

Because the analysis is path-insensitive, the flow information for analyzing s cannot know exactly the aliases to `x`. Therefore, the analysis rejects this program because `x` "escapes" before it is initialized.

Another possibility is to allow escaped uninitialized data. We could add an abstract rvalue to express that `y` points to uninitialized data without known exactly where it points. Assigning through `y` with initialized data could change `y` to be initialized. The Cyclone implementation used to have this extension, but the complexity does not seem worthwhile.

Path-insensitivity is not the only culprit for local variables escaping. If we assign `&x` to an unknown location (e.g., `*y` if we do not know exactly where `y` points), then `x` escapes. Also, if we pass `&x` to a function, then `x` escapes because the analysis is intraprocedural.

To dereference a possibly-`NULL` pointer in an escaped location, it is necessary to copy the pointer to an unescaped location and then test it:

```
void f(int *@x) {
  int *y = *x;
  if(y != NULL)
    *y = 123;
}
```

Copying is necessary so that an intervening assignment to the escaped location cannot compromise soundness. Making a copy is a well-known idiom for defensive programming; it is encouraging that the analysis enforces using it.

Path-insensitivity introduces approximations beyond causing locations to escape in the sense described above. The canonical example is data correlation between two if-statements, as in this example:

```
int f(int b, int *x) {
  if(b && x==NULL) return 0;
  if(b) return *x; // safe but rejected
  return 0;
}
```

It is not possible that `*x` dereferences `NULL` in this example, but the analysis rejects it because after the first if-statement, `x` might be `NULL`.

Interprocedural Idioms: Except for the extension described in Section 6.3.7, we do not allow passing uninitialized data to functions. Even this extension captures only the simplest such safe idioms.

As for `NULL` pointers, our only support for interprocedural idioms is τ@ and letting it be a subtype of $\tau*$. This subtyping does allow subtype polymorphism:

A function taking a parameter of type $\tau*$ can operate over data of type $\tau@$. In preceding chapters, we provided parametric polymorphism for features such as types, region names, and lock names. Indeed, subtype polymorphism has some weaknesses, as this example demonstrates:

```
int* f(int *p) {
  if(p != NULL)
    putc(*p);
  return p;
}
```

With "nullability polymorphism," we could express that the return type could be `NULL` if and only if the parameter could be. This equality lets callers assume the result is not `NULL` when the parameter is not `NULL`. Adding nullability polymorphism would create a more uniform type system, but it is unclear if the feature is necessary in practice.

Arrays: The shortcomings described so far are the more interesting ones from a technical point of view, but the most serious limitations in practice concern arrays. In short, arrays must be initialized when they are created and an array element has abstract value ALL$*$ unless it has some type $\tau@$. We do allow delaying the initialization of pointers to arrays; when they are initialized, they will refer to initialized arrays. To make initialization more palatable, Cyclone supports "comprehensions" (as in the example below and in C99 [107, 123]) and the argument to `new` can be an initializer. As Chapter 7 explains, the types for arrays and pointers to arrays include the size of the array. This silly example creates several arrays:

```
int f(int *x, int b) {
  int * arr1[37] = {for i < 37: x};
  int * arr2[23] = {for i < 23: arr1[i]};
  int **{23} p;
  if(b)
    p = arr2;
  else
    p = new {for i < 23: x};
  return p[14];
}
```

The first two declarations create stack-allocated arrays that are initialized with comprehensions. Within the body of the comprehension, a variable (`i` in our example) is bound to the index being initialized, so the second comprehension

copies a prefix of `arr1` into `arr2`. Comprehensions are often more convenient than initializers of the form {e_0,...,e_n}, and Cyclone prohibits omitting an array initializer. In C it is common to omit the initializer and use a for-loop (or a more complicated idiom) to initialize the array. It is also common to use `malloc` to create an array, which the analysis cannot support. The example also shows that pointers to arrays, such as `p`, can omit initializers.

Extending the flow analysis to reason about array indices in a useful and understandable way is difficult. If all index expressions (i.e., e_2 in $e_1[e_2]$) were compile-time constants, we could treat arrays just like `struct` values, but then there would be no need for arrays in our language. Allowing even slightly more complicated index expressions for uninitialized arrays is difficult. Consider this example:

```
int x[23];
for(int i=0; i < 23; ++i)
  x[i] = 37;
```
s

To conclude `x` is initialized before s, we need the correct loop invariant. Specifically, before entering the loop body, elements $0 \ldots (i-1)$ are initialized. Because the only control-flow path to s is from the test expression when $i \geq 23$, the loop invariant implies `x` is initialized. Automatic synthesis of such invariants for loops requires the analysis to incorporate a sound arithmetic. This dissertation does not investigate such extensions further. Instead, we resort to comprehensions, a special language construct that makes it almost trivial to ensure an array is initialized.

6.3.5 Example: Iterative List Copying

Consider this code for copying a list. (The syntax `new List(e1,e2)` heap-allocates a `struct List` and initializes the `hd` and `tl` fields to `e1` and `e2`, respectively.)

```
struct L { int hd; struct L *tl; };
struct L * copy(struct L * x) {
  struct L *result, *prev;              // line 1
  if (x == NULL) return NULL;           // line 2
  result = new List(x->hd,NULL);        // line 3
  prev = result;                        // line 4
  for (x=x->tl; x != NULL; x=x->tl) {   // line 5
    prev->tl = new List(x->hd,NULL);    // line 6
    prev = prev->tl;                    // line 7
  }                                     // line 8
  return result;                        // line 9
}
```

This example is not contrived. A polymorphic version is part of Cyclone's list library. It was written before Cyclone had support for static detection of `NULL`-pointer dereferences.

The analysis allows the dereferences of `x` (lines 3, 5, and 6) because they always follow explicit tests for `NULL` (lines 2 and 5) and there are no intervening assignments to `x`. More interestingly, the analysis does *not* allow the dereferences of `prev` (lines 6 and 7) without inserting implicit checks. We now describe how the iterative analysis reaches this conservative conclusion.

Let the allocation sites on lines 3 and 6 have names `a3` and `a6`, respectively. The abstract state (typing context) after analyzing line 4 maps `a3.hd` and `a3.tl` to ALL$*$ and `prev` and `result` to `&a3`. Therefore, after analyzing the loop body for the first time, the abstract state after line 7 maps `a3.tl` and `prev` to `&a6` and `result` to `&a3`. (It also maps `a6.hd` and `a6.tl` to ALL$*$.) We must iterate because the abstract rvalues for `prev` before and after the loop are incomparable.

To join the two abstract states, we make `prev` map to ALL@. (Doing so requires that the fields for `a3` and `a6` escape, so `a3.tl` maps to ALL@ in the joined state.) On the second iteration, the left-hand-side of the assignment on line 6 cannot dereference `NULL` (because `prev` maps to ALL@), but we are assigning `&a6` to an unknown location. Similarly, on line 7 the right-hand-side evaluates to the contents of an unknown location. Because `prev->tl` has type `struct L*`, the resulting abstract rvalue is ALL$*$. So after the assignment, `prev` maps to ALL$*$. Therefore, we must iterate again and consider both dereferences of `prev` potentially unsafe.

We have explained why the analysis rejects this code. The code is safe, be we can see why the analysis should reject this code: Suppose we inserted a function call `f(result)` between lines 6 and 7. This function could make `prev->tl` on line 7 evaluate to `NULL` (by using `result` to remove the last element of the list).

The following change allows the analysis to accept the function:

```
struct L * copy(struct L * x) {
  struct L *result, *prev;              // line 1
  if (x == NULL) return NULL;           // line 2
  result = new List(x->hd,NULL);        // line 3
  prev = result;                        // line 4
  for (x=x->tl; x != NULL; x=x->tl) {   // line 5
    struct L *tmp = new List(x->hd,NULL);
    prev->tl = tmp;                     // line 6
    prev = tmp;                         // line 7
  }                                     // line 8
  return result;                        // line 9
}
```

For the analysis, the difference is that the right-hand-side of line 7 on the second iteration abstractly evaluates to ALL@ instead of ALL∗. Intuitively, using `tmp` eliminates an implicit assumption that an escaped location is not mutated between lines 6 and 7.

6.3.6 Example: Cyclic Lists

We can use this `struct` type to implement nonempty doubly-linked cyclic lists of integers (where the "last" list element points to the "first" element):

```
struct CLst {
  int val;
  struct CLst @ prev;
  struct CLst @ next;
};
```

Because the `next` and `prev` fields cannot be `NULL`, code that traverses lists never needs to check for `NULL`. Functions for combining two cyclic lists and inserting a new element in a cyclic list are straightforward:

```
void append(struct CLst @lst1, struct CLst @lst2) {
  struct CLst @ n = lst1->next;
  struct CLst @ p = lst2->prev;
  lst1->next = lst2;
  lst2->prev = lst1;
  p->next = n;
  n->prev = p;
}

void insert(struct CLst @lst, int v) {
  struct CLst @ p = malloc(sizeof(struct CLst));
  p->val  = v;
  p->prev = lst->prev;
  p->next = lst;
  lst->prev = p;
  p->prev->next = p;
}
```

The interesting function is the one that creates a new single-element list:

```
struct CLst @ make(int v) {
  struct CLst @ ans = malloc(sizeof(struct CLst));
  ans->val  = v;
  ans->prev = ans;
  ans->next = ans;
  return ans;
}
```

Must points-to information is essential for accepting this function. Suppose our abstract rvalues did not include `&ans`. Now consider the assignment to `ans->prev`. The only remaining sound abstract rvalue for the right-hand side is NONE. Adding abstract rvalue describing pointers to partially initialized values does not help: `ans->prev=ans` makes `ans->prev` point to a value with an initialized `prev` field *only* because of aliasing.

This cyclic initialization problem is fairly well-known in the ML community. In ML, there is no way to create a cyclic list as defined above. Instead, it is necessary to use a datatype (i.e., make the `next` and `prev` fields possibly `NULL`) in order to create an initial cycle. Because the ML type system has no flow-sensitivity, every use of the fields must check that they refer to other list elements.

An alternative to flow-sensitive static checking is a special term form for creating and initializing cyclic data.

6.3.7 Constructor Functions

Intraprocedural analysis cannot support the idiom in which a caller passes a pointer to uninitialized data that the callee initializes. If the callee is `f`, we would like to allow `f(&x)` even if `x` is uninitialized. Furthermore, we would like to assume `x` is initialized after the call. I have implemented a somewhat ad hoc extension to Cyclone to support this idiom. (Although this idiom is common, it is unnecessary. We could change `f` to return an initialized object and replace `f(&x)` with `x=f()`.)

Because this idiom makes different interprocedural assumptions than other calls, we require an explicit annotation that changes the callee's type. The attribute `initializes(`i`)` indicates that a function initializes its i^{th} parameter. We can use this attribute only for parameters with types of the form τ@. This attribute changes how we analyze the caller and the callee.

For the callee, a parameter that it initializes starts with abstract value $\&x$ for some fresh x. Before any reachable control transfer to the caller (a `return` statement or the end of a function with return type `void`), we require x has abstract rvalue ALL$*$. This level of indirection is necessary because functions like the following must not type-check:

```
void f(int @p) attribute(initializes(1)) {
  int @ q = new 0;
  p = q;
  *p = 37; // does not initialize the correct memory
}
```

For the caller, it appears sound to allow abstract rvalues of the form $\&x$ or ALL@ for initialized parameters. For the former, we know x is initialized after the call. For the latter, the reinitialization is harmless.

Unfortunately, the typing rules just described have a subtle unsoundness: The callee assumes each parameter it initializes points to distinct memory. (It chooses a fresh variable name for each one.) Therefore, if a function initializes multiple arguments, we must forbid callers from passing the same location for these arguments. In general, the callee assumes x is unescaped, so we must enforce this fact at the call site. Therefore, we do not allow ALL@ for such parameters. In fact, we require distinct unescaped locations.

This support for constructor functions is limited. It does not support the idiom where the callee can return a value indicating whether or not it initialized a parameter. Another limitation is that callers cannot pass `NULL` for initialized parameters (to indicate they do not want a value). Supporting this idiom would require an abstract rvalue indicating, "`NULL` or $\&x$." Furthermore, it is unclear how to express what the abstract state must be before returning to the caller.

6.4 Formalism

This section develops an abstract machine for which uninitialized data and `NULL`-pointer dereferences make the machine stuck. A static semantics captures the key ideas of this chapter's flow analysis. Appendix D proves that programs well-formed according to this (unconventional) type system do not get stuck when they execute.

The greatest difference between the actual flow analysis and this chapter's formalization is that the formalism takes a declarative approach. It is neither syntax-directed nor iterative. As presented, a "type checker" would have to "guess" how to make abstract states more approximate to check loops. Nonetheless, it assigns an abstract state to each program point using flow-sensitive information. In Section 6.4.4, we sketch how to adjust the type system to make it more like a conventional flow analysis. Section 6.5 discusses advantages and disadvantages of formalizing the analysis as a type system.

$$
\begin{array}{rrcl}
\text{types} & \tau & ::= & \mathsf{int} \mid \tau* \mid \tau@ \\
\text{terms} & s & ::= & e \mid \mathsf{return} \mid \tau\ x \mid s; s \mid \mathsf{if}\ e\ s\ s \mid \mathsf{while}\ e\ s \\
 & e & ::= & i \mid \star \mid x \mid \&e \mid *e \mid e{=}e \mid \mathsf{junk} \mid e \| e \\
\text{values} & v & ::= & i \mid \&x \mid \mathsf{junk} \\
\text{heaps} & H & ::= & \cdot \mid H, x \mapsto v \\
\text{variable sets} & V & ::= & \cdot \mid V, x \\
\text{states} & P & ::= & V; H; s \\
\text{abstract rvalues} & r & ::= & \&x \mid 0 \mid \textsc{all@} \mid \textsc{all*} \mid \textsc{none} \\
\text{abstract lvalues} & \ell & ::= & x \mid ? \\
\text{escapednesses} & k & ::= & \textsc{unesc} \mid \textsc{esc} \\
\text{type contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau, k, r \\
\text{renamings} & M & ::= & \cdot \mid M, x \mapsto x
\end{array}
$$

Figure 6.1: Chapter 6 Formal Syntax

6.4.1 Syntax

Figure 6.1 presents the syntax for our formal language. Except for formalizing `NULL` pointers and uninitialized memory, it is much simpler than the formalisms in other chapters. In particular, there are no functions and no quantified types. Statements include expressions executed for their effect (e), a return statement that halts the program (return), memory allocation ($\tau\ x$), sequential composition ($s; s$), conditionals (if $e\ s\ s$), and loops (while $e\ s$). Allocation is like a variable declaration in C except that the memory lives forever. The memory initially holds junk; it must be initialized by an assignment expression. Because the variable can escape its scope, *even in the static semantics*, there is no reason to bind it in some enclosing statement. Rather, it is bound in the statement's continuation, e.g., s in $\tau\ x; s$, which is just sequential composition.

Expressions include integer constants (i), a nondeterministic form for producing an unknown integer ($\star$), variables (x), pointer creation ($\&e$), pointer dereference ($*e$), assignment ($e{=}e$), uninitialized data (junk), and a construct for evaluating expressions in an unspecified order ($e \| e$). Including $\star$ ensures no analysis can fully determine program behavior even though we have neither input nor functions. As in preceding chapters, we distinguish left-expressions (e in $\&e$ and $e{=}e'$) from right-expressions. The evaluation order for $e{=}e$ is unspecified as for $e \| e$, but the latter treats both expressions as right-expressions. As in conventional C implementations, we use the constant 0 for `NULL` pointers.

A heap maps variables to values. The junk value can have any type, but the

If ι is an element of any syntax class defined in Figure 6.1, then $\underline{\text{rename}(M, \iota)}$ is identical to ι except that for each $x \in \text{Dom}(M)$, every x contained in ι is replaced with $M(x)$.

$$\frac{}{\vdash_{\text{V}} e : \cdot} \qquad \frac{}{\vdash_{\text{V}} \mathsf{return} : \cdot} \qquad \frac{}{\vdash_{\text{V}} \tau\ x : \cdot, x}$$

$$\frac{\vdash_{\text{V}} s_1 : V_1 \quad \vdash_{\text{V}} s_2 : V_2 \quad V_1 \cap V_2 = \emptyset}{\begin{array}{l}\vdash_{\text{V}} s_1; s_2 : V_1 \cup V_2 \\ \vdash_{\text{V}} \mathsf{if}\ e\ s_1\ s_2 : V_1 \cup V_2\end{array}} \qquad \frac{\vdash_{\text{V}} s : V}{\vdash_{\text{V}} \mathsf{while}\ e\ s : V}$$

$$\frac{}{V \vdash_{\text{wf}} \cdot} \qquad \frac{y \notin V \quad V, y \vdash_{\text{wf}} M}{V \vdash_{\text{wf}} M, x \mapsto y}$$

Figure 6.2: Chapter 6 Semantics, Bindings and Renaming

static semantics ensures no well-typed program tries to dereference junk or use it for the test in a conditional. We consider heaps implicitly reorderable and treat them as partial maps as convenient.

We do not formalize aggregates, recursive types, or `malloc`. It is straightforward to do so, but such features significantly complicate the language and the soundness proof in unilluminating ways. (Section 6.4.3 explains one interesting complication `malloc` causes.)

Without aggregates or recursive types, the syntax for types is extremely simple. A pointer's type can indicate a not-`NULL` invariant ($\tau@$), else it has the form $\tau*$. As in preceding chapters, typing contexts (Γ) map variables to types. However, typing contexts also have flow-sensitive information described by abstract rvalues (r) and escapednesses (k). We consider typing contexts implicitly reorderable and treat them as partial maps as convenient.

The typing judgment for right-expressions produces an abstract rvalue that approximates every value to which the expression might evaluate at run-time. Briefly, $\&x$ describes pointers that must point to x; 0 describes only 0; ALL@ describes values that are not 0 and from which no sequence of pointer dereferences can produce junk; ALL$*$ describes values that may be 0 but from which junk is unreachable; and NONE describes all values, including junk.

Similarly, the typing judgment for left-expressions produces an abstract lvalue that approximates the variable to which the expression might evaluate at run-time. The x form means the expression must evaluate to x. Examples include the expression x and—assuming the typing context ensures y has abstract rvalue $\&x$—the expression $*y$. The other form is ?, which approximates all left-expressions.

Finally, if Γ indicates that x has escapedness UNESC, then all pointers to x

are known. More precisely, if the heap-typing judgment gives heap H the type Γ and $H(y) = \&x$, then the abstract rvalue for y in Γ must be $\&x$. The subtyping judgment on typing contexts enforces this property. On the other hand, if x has escapedness ESC, then its abstract rvalue must be ALL$*$ (or ALL@ if its type is some τ@). The well-formedness judgment on typing contexts enforces this property.

Because the formal type system tracks flow-sensitive information in a way that lets variables appear where a conventional type system would consider them out of scope, we do *not* allow implicit α-conversion.[2] Instead, the judgment $\vdash_{\text{V}} s : V$ ensures all allocations in s use distinct variables; V contains precisely these variables. For source programs, this property is straightforward, and the actual compiler achieves it internally by giving every allocation site a unique name. But in our formal dynamic semantics, it means the "unrolling" of a loop must change the bindings in one copy of the loop body. To do so, the machine state includes a set V of "used" names and the loop rule uses a mapping M to do the renaming. A mapping M is well-formed with respect to V (written $V \vdash_{\text{wf}} M$) if M is injective and does not map variables to elements of V. Figure 6.2 defines $\vdash_{\text{V}} s : V$ and $V \vdash_{\text{wf}} M$.

Although the formal semantics must carefully address these renaming issues, they are technical distractions that do not help explain the flow analysis. Readers should consider ignoring the uses of variable sets and just accept that the formalism handles variable clashes and systematic renaming.

6.4.2 Dynamic Semantics

The dynamic semantics is straightforward, so only a few interesting facts deserve mention. Rule DS6.1 allocates memory by extending the heap with a variable mapping to junk. Given return; s, the statement s is unreachable (see DS6.3), so s is irrelevant. Rules DS6.4 and DS6.5 indicate that the machine becomes stuck if a test expression is uninitialized. Rule DS6.6 uses renaming to ensure that the two copies of the loop body allocate different locations. The new bindings become part of the global set of "used" variables. In previous chapters, implicit α-conversion accomplished the same goal.

Rules DR6.6 and DR6.7 formalize the unspecified evaluation order for $e{=}e$ and $e\|e$. In particular, they do *not* require that "all of one" expression is evaluated before "all of the other." For example, given $(x{=}0\|x{=}1)\|x{=}2$, the heap could map x to 1, then to 0, and then to 2. Section 6.2 explains why this leniency would be much more problematic if our formal language had "sequential" expressions (such

[2] In actual Cyclone, more traditional type-checking precedes flow analysis, so strange scoping does not exist.

$$\frac{}{V; H; (\tau\ x) \xrightarrow{\text{s}} V; H, x \mapsto \mathsf{junk}; 0}\ \text{DS6.1}$$

$$\frac{}{V; H; (v; s) \xrightarrow{\text{s}} V; H; s}\ \text{DS6.2} \qquad \frac{}{V; H; (\mathsf{return}; s) \xrightarrow{\text{s}} V; H; \mathsf{return}}\ \text{DS6.3}$$

$$\frac{}{V; H; \mathsf{if}\ 0\ s_1\ s_2 \xrightarrow{\text{s}} V; H; s_2}\ \text{DS6.4} \qquad \frac{v \neq \mathsf{junk} \quad v \neq 0}{V; H; \mathsf{if}\ v\ s_1\ s_2 \xrightarrow{\text{s}} V; H; s_1}\ \text{DS6.5}$$

$$\frac{\vdash_{\text{v}} s : V_0 \quad \text{Dom}(M) = V_0 \quad V \vdash_{\text{wf}} M \quad s' = \text{rename}(M, s) \quad \vdash_{\text{v}} s' : V_1}{V; H; \mathsf{while}\ e\ s \xrightarrow{\text{s}} V \cup V_1; H; \mathsf{if}\ e\ (s'; \mathsf{while}\ e\ s)\ 0}\ \text{DS6.6}$$

$$\frac{H; e \xrightarrow{\text{r}} H'; e'}{\begin{array}{c} V; H; e \xrightarrow{\text{s}} V; H'; e' \\ V; H; \mathsf{if}\ e\ s_1\ s_2 \xrightarrow{\text{s}} V; H'; \mathsf{if}\ e'\ s_1\ s_2 \end{array}}\ \text{DS6.7} \qquad \frac{V; H; s \xrightarrow{\text{s}} V'; H'; s'}{V; H; (s; s_2) \xrightarrow{\text{s}} V'; H'; (s'; s_2)}\ \text{DS6.8}$$

Figure 6.3: Chapter 6 Dynamic Semantics, Statements

as C's `&&`, `||`, and comma operator). Rule DR6.5 makes the result of $e_1 \| e_2$ be the result of e_2. The semantics of expressions does not refer to V because expressions never allocate.

6.4.3 Static Semantics

Figure 6.5 defines several well-formedness judgments. Well-formed abstract lvalues must mention only variables in some assumed Γ. Well-formed abstract rvalues have the same restriction as well as restrictions regarding escapedness and types. If a variable has escapedness ESC, then its abstract rvalue is fixed: it must be ALL$*$ (or ALL@ if its type is τ@). This restriction is key for ensuring type preservation under assignment to escaped locations. We extend these restrictions to typing contexts with $\Gamma \vdash_{\text{wf}} \Gamma'$. A typing context Γ is well-formed and closed if $\Gamma \vdash_{\text{wf}} \Gamma$. Given just Γ, it is not possible to enforce that the escapedness information is sound, i.e., that there are no unknown pointers to variables with escapedness UNESC. Therefore, the abstract-ordering judgments enforce this property.

The judgment $V_1; V_2 \vdash_{\text{wf}} \Gamma$ is used as a technical restriction on what typing contexts the static semantics can "make up" for unreachable code. Type-checking unreachable code is rather pathological, but it is important for proving type preservation. It is technically convenient (but unnecessary for safety) to require the "made up" Γ to include every variable in V_2 and no variable not in V_1. Rules SS6.2 and ST6.1–3 use this judgment. In the actual flow-analysis algorithm, we do not make up typing contexts.

$$\frac{}{H; x \xrightarrow{\text{r}} H; H(x)}\ \text{DR6.1} \qquad \frac{}{H, x \mapsto v'; x{=}v \xrightarrow{\text{r}} H, x \mapsto v; v}\ \text{DR6.2}$$

$$\frac{}{H; *\&x \xrightarrow{\text{r}} H; x}\ \text{DR6.3} \qquad \frac{}{H; \star \xrightarrow{\text{r}} H; i}\ \text{DR6.4} \qquad \frac{}{H; v_1 \| v_2 \xrightarrow{\text{r}} H; v_2}\ \text{DR6.5}$$

$$\frac{H; e \xrightarrow{\text{r}} H'; e'}{\begin{array}{c} H; *e \xrightarrow{\text{r}} H'; *e' \\ H; e_1 \| e \xrightarrow{\text{r}} H'; e_1 \| e' \\ H; e \| e_2 \xrightarrow{\text{r}} H'; e' \| e_2 \\ H; e_1{=}e \xrightarrow{\text{r}} H'; e_1{=}e' \end{array}}\ \text{DR6.6} \qquad \frac{H; e \xrightarrow{\text{l}} H'; e'}{\begin{array}{c} H; \&e \xrightarrow{\text{r}} H'; \&e' \\ H; e{=}e_2 \xrightarrow{\text{r}} H'; e'{=}e_2 \end{array}}\ \text{DR6.7}$$

$$\frac{}{H; *\&x \xrightarrow{\text{l}} H; x}\ \text{DL6.1} \qquad \frac{H; e \xrightarrow{\text{r}} H'; e'}{H; *e \xrightarrow{\text{l}} H'; *e'}\ \text{DL6.2}$$

Figure 6.4: Chapter 6 Dynamic Semantics, Expressions

$$\frac{x \in \text{Dom}(\Gamma)}{\Gamma \vdash_{\text{wf}} x} \qquad \frac{}{\Gamma \vdash_{\text{wf}} ?}$$

$$\frac{}{\begin{array}{l} \Gamma \vdash_{\text{wf}} \text{int}, k, \textsc{all}* \\ \Gamma \vdash_{\text{wf}} \text{int}, \textsc{unesc}, \textsc{all}@ \\ \Gamma \vdash_{\text{wf}} \text{int}, \textsc{unesc}, 0 \\ \Gamma \vdash_{\text{wf}} \text{int}, \textsc{unesc}, \textsc{none} \end{array}} \qquad \frac{}{\begin{array}{l} \Gamma \vdash_{\text{wf}} \tau*, k, \textsc{all}* \\ \Gamma \vdash_{\text{wf}} \tau*, \textsc{unesc}, \textsc{all}@ \\ \Gamma \vdash_{\text{wf}} \tau*, \textsc{unesc}, 0 \\ \Gamma \vdash_{\text{wf}} \tau*, \textsc{unesc}, \textsc{none} \end{array}}$$

$$\frac{}{\begin{array}{l} \Gamma \vdash_{\text{wf}} \tau@, k, \textsc{all}@ \\ \Gamma \vdash_{\text{wf}} \tau@, \textsc{unesc}, \textsc{none} \end{array}} \qquad \frac{x \in \text{Dom}(\Gamma)}{\begin{array}{l} \Gamma \vdash_{\text{wf}} \tau*, \textsc{unesc}, \&x \\ \Gamma \vdash_{\text{wf}} \tau@, \textsc{unesc}, \&x \end{array}}$$

$$\frac{}{\Gamma \vdash_{\text{wf}} \cdot} \qquad \frac{\Gamma \vdash_{\text{wf}} \Gamma' \quad \Gamma \vdash_{\text{wf}} \tau, k, r}{\Gamma \vdash_{\text{wf}} \Gamma', x{:}\tau, k, r}$$

$$\frac{\Gamma \vdash_{\text{wf}} \Gamma \quad V_2 \subseteq \text{Dom}(\Gamma) \subseteq V_1 \cup V_2}{V_1; V_2 \vdash_{\text{wf}} \Gamma}$$

Figure 6.5: Chapter 6 Well-Formedness

$$\frac{}{\vdash k \leq k} \qquad \frac{}{\vdash \textsc{unesc} \leq \textsc{esc}}$$

$$\frac{}{\Gamma \vdash \ell \leq \ell} \qquad \frac{}{\Gamma, x{:}\tau, \textsc{esc}, r \vdash x \leq ?}$$

$$\frac{}{\Gamma \vdash r \leq r} \qquad \frac{\Gamma \vdash r_1 \leq r_2 \quad \Gamma \vdash r_2 \leq r_3}{\Gamma \vdash r_1 \leq r_3}$$

$$\frac{}{\Gamma, x{:}\tau, \textsc{esc}, r \vdash \&x \leq \textsc{all@}}$$

$$\frac{}{\Gamma \vdash 0 \leq \textsc{all*}} \qquad \frac{}{\Gamma \vdash \textsc{all@} \leq \textsc{all*}} \qquad \frac{}{\Gamma \vdash r \leq \textsc{none}}$$

$$\frac{}{\Gamma \vdash \cdot \leq \cdot} \qquad \frac{\Gamma \vdash \Gamma_1 \leq \Gamma_2 \quad \vdash k_1 \leq k_2 \quad \Gamma \vdash r_1 \leq r_2}{\Gamma \vdash \Gamma_1, x{:}\tau, k_1, r_1 \leq \Gamma_2, x{:}\tau, k_2, r_2}$$

Figure 6.6: Chapter 6 Abstract Ordering

$$\frac{\Gamma \vdash_{\text{rtyp}} e : \tau, r, \Gamma'}{V; \Gamma \vdash_{\text{styp}} e : \Gamma'} \text{ SS6.1} \qquad \frac{V; \text{Dom}(\Gamma) \vdash_{\text{wf}} \Gamma'}{V; \Gamma \vdash_{\text{styp}} \mathsf{return} : \Gamma'} \text{ SS6.2}$$

$$\frac{V; \Gamma \vdash_{\text{styp}} s_1 : \Gamma'' \quad V - V_1; \Gamma'' \vdash_{\text{styp}} s_2 : \Gamma' \quad \vdash_{\text{V}} s_1 : V_1}{V; \Gamma \vdash_{\text{styp}} s_1; s_2 : \Gamma'} \text{ SS6.3}$$

$$\frac{V; \Gamma \vdash_{\text{tst}} e : \Gamma_1; \Gamma_2 \quad V; \Gamma_1 \vdash_{\text{styp}} s : \Gamma}{V; \Gamma \vdash_{\text{styp}} \mathsf{while}\ e\ s : \Gamma_2} \text{ SS6.4}$$

$$\frac{V; \Gamma \vdash_{\text{tst}} e : \Gamma_1; \Gamma_2 \quad V; \Gamma_1 \vdash_{\text{styp}} s_1 : \Gamma' \quad V; \Gamma_2 \vdash_{\text{styp}} s_2 : \Gamma'}{V; \Gamma \vdash_{\text{styp}} \mathsf{if}\ e\ s_1\ s_2 : \Gamma'} \text{ SS6.5}$$

$$\frac{}{V; \Gamma \vdash_{\text{styp}} \tau\ x : \Gamma, x{:}\tau, \textsc{unesc}, \textsc{none}} \text{ SS6.6}$$

$$\frac{V; \Gamma_0 \vdash_{\text{styp}} s : \Gamma_1 \quad \Gamma_2 \vdash \Gamma_1 \leq \Gamma_2 \quad \Gamma_2 \vdash_{\text{wf}} \Gamma_2}{V; \Gamma_0 \vdash_{\text{styp}} s : \Gamma_2} \text{ SS6.7}$$

$$\frac{V; \Gamma_0 \vdash_{\text{styp}} s : \Gamma_1 \Gamma_2 \quad \Gamma_2 \vdash_{\text{wf}} \Gamma_2}{V; \Gamma_0 \vdash_{\text{styp}} s : \Gamma_2} \text{ SS6.8}$$

Figure 6.7: Chapter 6 Typing, Statements

$$\frac{}{\Gamma \vdash_{\text{rtyp}} \mathsf{junk} : \tau, \text{NONE}, \Gamma}\ \text{SR6.1} \qquad \frac{}{\Gamma \vdash_{\text{rtyp}} 0 : \tau*, 0, \Gamma}\ \text{SR6.2} \qquad \frac{}{\Gamma \vdash_{\text{rtyp}} 0 : \text{int}, 0, \Gamma}\ \text{SR6.3}$$

$$\frac{i \neq 0}{\Gamma \vdash_{\text{rtyp}} i : \text{int}, \text{ALL@}, \Gamma}\ \text{SR6.4} \qquad \frac{}{\Gamma \vdash_{\text{rtyp}} \star : \text{int}, \text{ALL*}, \Gamma}\ \text{SR6.5} \qquad \frac{\Gamma(x) = \tau, k, r}{\Gamma \vdash_{\text{rtyp}} x : \tau, r, \Gamma}\ \text{SR6.6}$$

$$\frac{\Gamma \vdash_{\text{ltyp}} e : \tau, x, \Gamma'}{\Gamma \vdash_{\text{rtyp}} \&e : \tau@, \&x, \Gamma'}\ \text{SR6.7A} \qquad \frac{\Gamma \vdash_{\text{ltyp}} e : \tau, ?, \Gamma'}{\Gamma \vdash_{\text{rtyp}} \&e : \tau@, \text{ALL@}, \Gamma'}\ \text{SR6.7B}$$

$$\frac{\Gamma \vdash_{\text{rtyp}} e : \tau*, \&x, \Gamma' \quad \Gamma'(x) = \tau, k, r}{\Gamma \vdash_{\text{rtyp}} *e : \tau, r, \Gamma'}\ \text{SR6.8A} \qquad \frac{\Gamma \vdash_{\text{rtyp}} e : \tau@*, \text{ALL@}, \Gamma'}{\Gamma \vdash_{\text{rtyp}} *e : \tau@, \text{ALL@}, \Gamma'}\ \text{SR6.8B}$$

$$\frac{\Gamma \vdash_{\text{rtyp}} e : \tau**, \text{ALL@}, \Gamma'}{\Gamma \vdash_{\text{rtyp}} *e : \tau*, \text{ALL*}, \Gamma'}\ \text{SR6.8C} \qquad \frac{\Gamma \vdash_{\text{rtyp}} e : \text{int}*, \text{ALL@}, \Gamma'}{\Gamma \vdash_{\text{rtyp}} *e : \text{int}, \text{ALL*}, \Gamma'}\ \text{SR6.8D}$$

$$\frac{\Gamma \vdash_{\text{rtyp}} e_1 : \tau', r', \Gamma \quad \Gamma \vdash_{\text{rtyp}} e_2 : \tau, r, \Gamma}{\Gamma \vdash_{\text{rtyp}} e_1 \| e_2 : \tau, r, \Gamma}\ \text{SR6.9}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\text{ltyp}} e_1 : \tau_1, \ell, \Gamma \\ \Gamma \vdash_{\text{rtyp}} e_2 : \tau_2, r, \Gamma \\ \Gamma \vdash_{\text{aval}} \tau_1, \ell, r, \Gamma' \\ \vdash_{\text{atyp}} \tau_1, \tau_2, r\end{array}}{\Gamma \vdash_{\text{rtyp}} e_1 = e_2 : \tau_2, r, \Gamma'}\ \text{SR6.10} \qquad \frac{}{\vdash_{\text{atyp}} \tau, \tau, r} \qquad \frac{\Gamma \vdash_{\text{wf}} \tau, \text{ESC}, r}{\Gamma \vdash_{\text{aval}} \tau, ?, r, \Gamma}$$

$$\frac{r \neq \text{NONE}}{\vdash_{\text{atyp}} \tau@, \tau*, r} \qquad \frac{\Gamma, x{:}\tau, k, r \vdash_{\text{wf}} \tau, k, r}{\Gamma, x{:}\tau, k, r' \vdash_{\text{aval}} \tau, x, r, \Gamma, x{:}\tau, k, r}$$

$$\frac{\Gamma \vdash_{\text{rtyp}} e : \tau@, r, \Gamma'}{\Gamma \vdash_{\text{rtyp}} e : \tau*, r, \Gamma'}\ \text{SR6.11} \qquad \frac{\begin{array}{l}\Gamma_0 \vdash_{\text{rtyp}} e : \tau, r, \Gamma_1 \\ \Gamma_2 \vdash \Gamma_1 \leq \Gamma_2 \\ \Gamma_2 \vdash_{\text{wf}} \Gamma_2\end{array}}{\Gamma_0 \vdash_{\text{rtyp}} e : \tau, r, \Gamma_2}\ \text{SR6.12} \qquad \frac{\begin{array}{l}\Gamma_0 \vdash_{\text{rtyp}} e : \tau, r', \Gamma_1 \\ \Gamma_1 \vdash r' \leq r\end{array}}{\Gamma_0 \vdash_{\text{rtyp}} e : \tau, r, \Gamma_1}\ \text{SR6.13}$$

$$\frac{\Gamma(x) = \tau, k, r}{\Gamma \vdash_{\text{ltyp}} x : \tau, x, \Gamma}\ \text{SL6.1} \qquad \frac{\Gamma \vdash_{\text{rtyp}} e : \tau*, \&x, \Gamma'}{\Gamma \vdash_{\text{ltyp}} *e : \tau, x, \Gamma'}\ \text{SL6.2A} \qquad \frac{\Gamma \vdash_{\text{rtyp}} e : \tau*, \text{ALL@}, \Gamma'}{\Gamma \vdash_{\text{ltyp}} *e : \tau, ?, \Gamma'}\ \text{SL6.2B}$$

$$\frac{\begin{array}{l}\Gamma_0 \vdash_{\text{ltyp}} e : \tau, \ell, \Gamma_1 \\ \Gamma_2 \vdash \Gamma_1 \leq \Gamma_2 \\ \Gamma_2 \vdash_{\text{wf}} \Gamma_2\end{array}}{\Gamma_0 \vdash_{\text{ltyp}} e : \tau, \ell, \Gamma_2}\ \text{SL6.3} \qquad \frac{\begin{array}{l}\Gamma_0 \vdash_{\text{ltyp}} e : \tau, \ell', \Gamma_1 \\ \Gamma_1 \vdash \ell' \leq \ell\end{array}}{\Gamma_0 \vdash_{\text{ltyp}} e : \tau, \ell, \Gamma_1}\ \text{SL6.4}$$

Figure 6.8: Chapter 6 Typing, Expressions

$$\frac{\Gamma \vdash_{\text{rtyp}} e : \tau, 0, \Gamma_2 \quad V; \text{Dom}(\Gamma) \vdash_{\text{wf}} \Gamma_1}{V; \Gamma \vdash_{\text{tst}} e : \Gamma_1; \Gamma_2} \text{ ST6.1}$$

$$\frac{\Gamma \vdash_{\text{rtyp}} e : \tau, \&x, \Gamma_1 \quad V; \text{Dom}(\Gamma) \vdash_{\text{wf}} \Gamma_2}{V; \Gamma \vdash_{\text{tst}} e : \Gamma_1; \Gamma_2} \text{ ST6.2}$$

$$\frac{\Gamma \vdash_{\text{rtyp}} e : \tau, \text{ALL@}, \Gamma_1 \quad V; \text{Dom}(\Gamma) \vdash_{\text{wf}} \Gamma_2}{V; \Gamma \vdash_{\text{tst}} e : \Gamma_1; \Gamma_2} \text{ ST6.3}$$

$$\frac{\Gamma \vdash_{\text{ltyp}} e : \tau, x, \Gamma', x{:}\tau, \text{UNESC}, \text{ALL}*}{V; \Gamma \vdash_{\text{tst}} e : (\Gamma', x{:}\tau, \text{UNESC}, \text{ALL@}); (\Gamma', x{:}\tau, \text{UNESC}, 0)} \text{ ST6.4}$$

$$\frac{\Gamma \vdash_{\text{rtyp}} e : \tau, \text{ALL}*, \Gamma_1}{V; \Gamma \vdash_{\text{tst}} e : \Gamma_1; \Gamma_1} \text{ ST6.5}$$

Figure 6.9: Chapter 6 Typing, Tests

$$\frac{}{\Gamma \vdash_{\text{htyp}} \cdot : \cdot} \qquad \frac{\Gamma \vdash_{\text{htyp}} H : \Gamma' \quad \Gamma \vdash_{\text{rtyp}} v : \tau, r, \Gamma}{\Gamma \vdash_{\text{htyp}} H, x \mapsto v : \Gamma', x{:}\tau, k, r}$$

$$\frac{\begin{array}{ll} \Gamma \vdash_{\text{htyp}} H : \Gamma & \vdash_{\text{V}} s : V'' \qquad V'' \subseteq V' \\ V'; \Gamma \vdash_{\text{styp}} s : \Gamma' & V' \cap \text{Dom}(H) = \emptyset \\ \Gamma \vdash_{\text{wf}} \Gamma & V \supseteq V' \cup \text{Dom}(H) \end{array}}{\vdash_{\text{prog}} V; H; s : \Gamma'}$$

Figure 6.10: Chapter 6 Typing, Program States

Figure 6.6 defines the abstract-ordering judgments, which formalize how we can lose information by choosing more approximate flow information. First, $\vdash k_1 \leq k_2$ indicates that locations can escape (though the associated abstract rvalue may need to change for the result to be well-formed). The judgment $\Gamma \vdash \ell_1 \leq \ell_2$ lets us forget a variable (which is a left-value), but *only* if x has escaped. Similarly, $\Gamma \vdash r_1 \leq r_2$ lets us forget that a value is initialized and forget that a value is or is not 0. We can forget must points-to information, but *only* if the pointed-to location has escaped. The last judgment has the form $\Gamma \vdash \Gamma_1 \leq \Gamma_2$, indicating that under the assumptions in Γ, we can approximate Γ_1 with Γ_2. The rules let us extend the other ordering judgments point-wise through Γ_1, but they do not imply that Γ_2 is well-formed.

Statements: Figure 6.7 presents the typing rules for statements. Because typing contexts describe flow-sensitive information, statements (and expressions) type-check under one typing context and produce another typing context. In $V; \Gamma \vdash_{\text{styp}} s : \Gamma'$, the resulting context is Γ'. Rule SS6.1 uses the typing judgment for right-expressions, explained below. Rule SS6.2 formalizes the notion that there is no control flow after a return, so it is sound to produce any Γ'. We impose technical restrictions on Γ' that simplify the safety proof.

Rules SS6.3–SS6.5 demonstrate how we reuse typing contexts to describe control flow. For $s_1; s_2$, control flows from s_1 to s_2, so SS6.3 uses the same Γ'' for the context produced by s_1 and the context assumed by s_2. For the test expressions in conditionals and loops, we use a typing judgment (explained below) that produces two typing contexts, one for when the test is not 0 and one for when it is 0. In rule SS6.5, we use one context for s_1 and the other for s_2. Both s_1 and s_2 must produce the same result context. (Rules SS6.7 and SS6.8 provides enough subsumption that equality is not overly restrictive.) For rule SS6.4, the resulting type context is the same as the false context from the test because only this control flow terminates the loop.

The strange variable sets in SS6.3 ensure s_2 does not make up variables that s_1 might allocate. We particularly do want to allow this behavior in SS6.5 so that the typing context produced for statements like if e (τ x) return can mention x.

Rule SS6.6 formalizes the fact that memory allocation produces fresh, uninitialized memory for which all aliases are known. It does not apply if $x \in \text{Dom}(\Gamma)$. (This treatment differs from the algorithm in Section 6.2, which keeps all allocation sites in the abstract state. Extending Γ is simpler in a declarative system. The implementation also extends abstract states in this way, but is equivalent to keeping all locations in all abstract states.)

The subsumption rule SS6.7 lets us produce a more conservative typing context.

However, this rule does not let us forget that bindings exist, which is necessary if a loop body or conditional branch allocates memory. Rule SS6.8 lets us restrict the domain of Γ', if the result is well-formed. It is tempting to make domain restriction part of abstract ordering, such as with this rule:

$$\frac{\Gamma \vdash \Gamma_1 \leq \Gamma_2}{\Gamma \vdash \Gamma_1, x{:}\tau, k, r \leq \Gamma_2}$$

However, if Γ_1 is $x{:}\tau@, \textsc{unesc}, \&y, y{:}\tau, \textsc{unesc}, \textsc{all}*$ and Γ_2 is $x{:}\tau@, \textsc{unesc}, \textsc{all}@$, we cannot show $\Gamma_2 \vdash \Gamma_1 \leq \Gamma_2$ because $y \notin \mathrm{Dom}(\Gamma_2)$. We lose no expressive power by restricting the domain only *after* using more approximate abstract rvalues and escapednesses.

Expressions: Figure 6.8 presents two interdependent typing judgments for expressions. The judgment for right-expressions has the form $\Gamma \vdash_{\text{rtyp}} e : \tau, r, \Gamma'$ because a right-expression has a type, an abstract rvalue approximating all values to which e might evaluate, and an effect such that given Γ, e produces Γ'. Similarly, the typing judgment for left-expressions concludes a type, an abstract lvalue, and a typing context. Although it appears that the rules never make explicit use of the escapedness information in typing contexts, well-formedness and abstract-ordering hypotheses use this information. We have more rules than in conventional type systems because the appropriate rule may depend on various abstract rvalues and not just the syntax of the term being type-checked.

Rule SR6.1–SR6.6 type-check effect-free expressions, so they produce the same Γ they consume. The constant 0 is an integer and a possibly-`NULL` pointer, so we have two rules for it. In both cases, the abstract rvalue is 0. A nonzero integer has type int and abstract rvalue $\textsc{all}@$. Similarly, $\star$ evaluates to an initialized integer that may be 0. For SR6.6, the type and abstract rvalue is in Γ.

Rules SR6.6A and SR6.7B type-check expressions of the form $\&e$. Such expressions evaluate to (nonzero) pointers. If e must left-evaluate to some location x, then $\&e$ must evaluate to $\&x$. Otherwise, the typing rules for left-expressions ensure $\textsc{all}@$ is appropriate.

The rules for $*e$ (SR6.8A–D) let us exploit a range of information from type-checking e. The most information we could we have is that e points to some location x, in which case $\Gamma'(x)$ holds an appropriate abstract rvalue. If we do not know where e points, then we require that e is neither `junk` nor 0, so we require that its abstract rvalue is $\textsc{all}@$. The abstract rvalue of $*e$ then depends on the type of e: It might be 0 (as $\textsc{all}*$ indicates) unless its type indicates otherwise (rule SR6.8B). We do not need rules where the type of e has the form $\tau'@$ because rule SR6.11 provides the appropriate subtyping.

The remaining expression forms are $e_1 \| e_2$ and $e_1 = e_2$, for which the dynamic semantics does not specify the order of evaluation. For reasons we explain in Section 6.2, it is sound to require that neither e_1 nor e_2 affects the abstract flow information (but it would not be if we had sequential expressions). Therefore, we require not only that e_1 and e_2 type-check, but that they produce the same Γ they consume. For rule SR6.9, this technique is the only interesting feature.

For SR6.10, we must ensure the assignment is safe and use it to produce the resulting flow information. We use the auxiliary $\vdash_{\text{atyp}}$ and $\vdash_{\text{aval}}$ judgments to avoid having four assignment rules. The purpose of $\vdash_{\text{atyp}}$ is to disallow assigning integers to pointers or vice-versa. We allow assigning $\tau*$ to $\tau@$ if the value is neither `junk` nor 0 ($\vdash_{\text{aval}}$ ensures the latter) and vice-versa (using rule SR6.11). If ℓ is ? or some escaped x, then r must be appropriate for an escaped location, as the well-formedness hypotheses in the $\vdash_{\text{aval}}$ rules enforce. For any type, there is only one such r, so the flow information cannot actually change. If ℓ is some unescaped x, then the assignment can change the flow information.

Rule SR6.10 would be too weak to support `malloc` because assignments of the form e`=malloc(sizeof(`τ`))` could not add an allocation site to the context. It suffices to add a rule for this case because the memory allocation is safe regardless of evaluation order. A special rule is unnecessary using the "changed sets" approach described in Section 6.2.

Rules SR6.11–13 provide subtyping. Rule SR6.11 lets us treat nonzero pointers as possibly-zero pointers. Such a rule is unsound for left-expressions. Rules SR6.12 and SR6.13 let us conclude a more approximate Γ and r respectively. Such subsumption may be necessary for expressions with undefined evaluation order (so that the flow information does not change) and for assignment to escaped locations (so that the Γ produced is well-formed). Because expressions do not allocate, it is never necessary to restrict the domain of a typing context. As a result, some lemmas in Appendix D are simpler for expressions than for statements.

The rules for typing left-expressions are straightforward adaptations of similar rules for right-expressions. As with right-expressions, we cannot dereference values that might be 0 or `junk`. Although subsumption is not useful for source programs, it helps establish type preservation when the abstract machine takes a step using rule DL6.1.

Tests: The typing rules for conditionals and while loops use the judgment $V; \Gamma \vdash_{\text{tst}} e : \Gamma_1; \Gamma_2$, which Figure 6.9 defines. We use the judgment to ensure Γ_1 and Γ_2 are sound approximations assuming e evaluates to a nonzero value and 0, respectively. If we can determine the zeroness of e statically, then one of Γ_1 or Γ_2 is irrelevant because the statements we type-check under the context will never be executed.

This fact explains rules ST6.1–3: One typing context is "made up." Rule ST6.4 lets us refine the abstract rvalue for an unescaped location. The rule formalizes the intuition that if an expression with abstract rvalue ALL$*$ is not zero (respectively, is zero), then it can have abstract rvalue ALL@ (respectively, 0). Rule ST6.5 addresses the case where the test cannot affect the flow information.

States: Finally, we type-check program states with the two judgments in Figure 6.10. The rules for type-checking heaps are what we would expect. To type-check a program state $V; H; s$, we type-check s under a context Γ that describes H. We also require that Γ is well-formed (so escaped locations have appropriate abstract rvalues). Although it may be possible to define an algorithm that finds the least approximate Γ such that $\Gamma \vdash_{\text{htyp}} H : \Gamma$ (or determine that no such Γ exists), we have no reason to do so because in practice we check only source programs. (The same is true in earlier chapters, but there the type-checking rules for heaps are essentially syntax-directed.)

The other hypotheses for $\vdash_{\text{prog}}$ are technical conditions to control renaming. The allocations in s must use distinct variables that are not already in the heap. At run-time, the dynamic semantics uses V to avoid reusing variables, so V must subsume variables in H and in s.

For a source program, $\vdash_{\text{prog}}$ amounts to ensuring the program type-checks under an empty typing context and does not have name clashes.

Summary: We have used type-theoretic techniques to specify a static semantics that incorporates flow-sensitive information including must points-to information. Several "tricks" deserve mentioning again. First, we express possible control flow by using the same typing context Γ multiple times in a rule. Second, we ensure the must points-to information is sound by allowing the flow information to determine that some location x points to another location y only when all aliases of x are known. Third, we use subsumption on typing contexts to specify an abstract-ordering relationship. The subsumption rules contain the part of the type system that does not lead directly to an algorithm. Fourth, we allow appropriate test expressions to refine flow information. Fifth, because the analysis does little to restrict variables' scope, we disallow implicit α-conversion.

6.4.4 Iterative Algorithm

This section describes how the formal declarative type system differs from the iterative flow analysis and how we might reduce the differences. Most importantly, the formal system allows more approximate abstract states "anywhere" (see rules

SS6.7–8, SR6.11–13, SL6.3–4) whereas the iterative analysis uses a particular join operation only when a program point has multiple control-flow predecessors. Well-known results in flow analysis suggest that the iterative analysis does not lose expressive power from this restriction.

It would be straightforward to enforce a similar restriction in our formal system. Essentially, we would remove the subsumption rules from the static semantics and modify rules for terms with multiple control-flow predecessors to use the join operator. For loops, we need a fixpoint operator (i.e., iteration) over the join. Because our approach to under-specified evaluation order is like assuming expressions may execute multiple times, we would use the fixpoint operator with SR6.9 and SR6.10 too.

Instead of making up typing contexts (see SS6.2 and ST6.1–3), the iterative algorithm produces an explicit $\bot$. Adding $\bot$ to our formalism is straightforward. The essential additions are the axioms $\Gamma \vdash_{\text{wf}} \bot$, $\Gamma \vdash \bot \leq \Gamma'$, $V; \bot \vdash_{\text{styp}} e : \bot$, and $V; \bot \vdash_{\text{tst}} e : \bot; \bot$.

We can easily dismiss other sources of nondeterminism. For tests, we can use ST6.4 only when ST6.1–3 do not apply and ST6.5 only when ST6.1–4 do not apply. For assigning a type to 0 or subsuming $\tau@$ to $\tau*$, we recall that in the Cyclone implementation, type-checking precedes flow analysis. This earlier compiler phase assigns types to all expressions. This dissertation does not discuss the details of subtyping or type inference.

One technical point does make the declarative type system more powerful than the iterative analysis: If our join operation needs to make the abstract rvalue $\&x$ more approximate, it always chooses ALL@ or ALL* and makes x escaped, failing if x is uninitialized. Our static semantics lets us replace $\&x$ with NONE and leave x unescaped. As such, our type-safety result implies a join operation more flexible in this regard remains sound.

Although the iterative flow analysis handles unstructured control flow naturally, it is awkward to extend our declarative system for it. The static semantics for `goto L` and `L:` s is no problem: Our context could include a map from labels to abstract states. If `L` mapped to Γ, then the abstract state at `goto L` would have to approximate Γ and `L:` s would have to check under Γ. As expected, the formalism "guesses" the mapping that the flow analysis discovers iteratively. However, our *dynamic* semantics would require substantial modification to support `goto`. Local term rewriting no longer suffices. A lower-level view of execution with an explicit "program counter" in the machine state should suffice. Some work discussed in Chapter 8, particularly Typed Assembly Language [157], takes this approach.

6.4.5 Type Safety

Appendix D proves this result:

Definition 6.1. *State $V; H; s$ is stuck if s is not some value v, s is not* return*, and there are no V', H' and s' such that $V; H; s \xrightarrow{s} V'; H'; s'$.*

Theorem 6.2 (Type Safety). *If $V; \cdot \vdash_{\text{styp}} s : \Gamma$, $\vdash_{\text{V}} s : V$, and $V; \cdot; s \xrightarrow{s}{}^{*} V'; H'; s'$ (where $\xrightarrow{s}{}^{*}$ is the reflexive transitive closure of $\xrightarrow{s}$), then $V'; H'; s'$ is not stuck.*

The proof is surprisingly difficult. Omitting pairs, recursive types, sequential expressions, and unstructured control flow from our formalism allows us to focus on the essential properties. Because the machine cannot dereference 0 or junk, the theorem implies we prevent such operations.

6.5 Related Work

Flow-sensitive information is a mainstay of modern compilation and program analysis. Most compiler textbooks explain how to define and implement dataflow analyses over intermediate representations of source programs [2, 158, 8]. Such analyses enjoy well-understood mathematical foundations that support their correctness and efficient implementation [166]. In this section, we discuss only work related to the more unusual features of this chapter's flow analysis. These features include the following:

- The analysis is for a general-purpose source language and is part of the language's definition.
- The analysis statically prevents safety violations resulting from using uninitialized data and dereferencing `NULL` pointers.
- The analysis is for a language with under-specified evaluation order.
- The analysis incorporates must points-to information.
- The formalism for the analysis uses type-theoretic techniques even though it describes flow-sensitive information.

Source-Language Flow Analysis The Java definition [92] requires implementations to enforce a particular compile-time intramethod flow analysis. This analysis prevents reading uninitialized local variables and prevents value-returning methods from reaching the end of the method body. The analysis interprets test-expressions accurately enough to accept methods like the following:

```
int f1() { while(true) ; }
void f2(int z) { int x; int y; if(((x=3) == z) && (y=x)) f2(x+y); }
```

The widespread use of Java is evidence that a general-purpose programming language can effectively include a conservative flow analysis in its definition. If the analysis supports enough common idioms, then programmers have little need to learn the specific rules. For example, they can omit initializers as they wish and use any resulting error messages to guide the introduction of initializers. I suspect that programmers use such interaction to gain an approximate understanding of the analysis that satisfies their needs.

The flow analyses for Java and Cyclone are quite similar because the former served as an inspiration and a starting point for the latter. The Java analysis is much simpler for several reasons. First, only a method's local variables can be uninitialized. Object fields (including array elements) and class (static) fields are implicitly initialized with default values (0 or `NULL`) before the object's constructor is called or the class is initialized, respectively. This decision avoids difficult interactions with subclassing and the order that constructors get called.

Second, there is no address-of operator. Together with the previous reason, this fact means there are never pointers (references) to uninitialized memory. In Cyclone terms, the form of Java's abstract rvalues is just ALL and NONE. Possibly-uninitialized locations cannot escape, so we do not need escapedness. The analysis rules for left-expressions also become much simpler: a variable x has abstract lvalue x whereas all other left-expressions have abstract lvalue ?.

Third, Java's analysis prohibits all uses of possibly-uninitialized locations. That is, if x has abstract rvalue NONE, then x can appear only an expression of the form $x{=}e$. Cyclone is more permissive. For example, we allow $y{=}x$ so long as y is unescaped. The abstract effect is to change the abstract rvalue in y to NONE. In Java, an initialized location can never become uninitialized. In terms of Cyclone's formalism, the typing context that a Java expression produces is never more approximate than the typing context it consumes. Allowing $y{=}x$ in Cyclone where x may be uninitialized is not very useful. However, it is important to allow $y{=}x$ where y contains abstract rvalue ALL@ and x contains ALL$*$. Such an assignment produces a more approximate typing context.

Fourth, Java does not have `goto`. Unlike Cyclone, if a statement s is unreachable, then every statement contained in s is unreachable. Therefore, we can conservatively determine reachability in a Java method body with one top-down pass over the code.

Fifth, evaluation order in Java is deterministic.

It turns out that the above reasons simplify Java's analysis so much that an algorithmic implementation has no need to iterate. Expressions have only one exe-

cution order, so the iterative "join approach" developed in this chapter is unnecessary. For statements, iteration is necessary only if a control transfer's destination has already been analyzed under a less approximate context than the control transfer's source. For Java, if we analyze method bodies "top to bottom," then control transfers to already analyzed statements can arise only from `continue` and reaching the end of the loop body. In both cases, the source's context is less approximate because variables cannot become uninitialized after they are initialized.

The fact that locations stay initialized also simplifies the analysis of exception handlers. In Cyclone, it is necessary to analyze a catch-clause under every typing context encountered in the corresponding try-body (except program points contained in a nested exception handler). In Java, it suffices to analyze the catch-clause under the initial typing context for the try-body.

Finally, Java does not have aggregate values (only pointers to aggregate values), so we have no need for abstract rvalues of the form $r \times r$. Put another way, it is impossible to initialize part of an uninitialized variable.

Static Control NULL-Pointer Checking Unlike memory initialization, Java always considers `NULL`-pointer dereferences a run-time error. Implementations may use static analysis to omit unnecessary checks for `NULL`, but there is no way for programmers to express a not-`NULL` invariant as with Cyclone's τ@ types.

Several research projects have explored tools and languages that provide this ability. ESC/Java [76] and Splint [189] provide annotations indicating that a function parameter or object field has a pointer that must never be `NULL`. These systems check such assertions at compile-time, subject to the soundness restrictions described in Chapter 8. These systems can also warn about dereferences of possibly-`NULL` pointers. Because these systems are tools, there is less concern about defining (in terms of the programming language) a precise notion of what restrictions they enforce.

Fähndrich and Leino [67] investigate retrofitting the safe object-oriented languages Java and C#, with not-`NULL` types. The main complication is object fields and array elements with not-`NULL` types. Object fields are initialized to `NULL` at run-time. To ensure they do not remain `NULL`, Fähndrich and Leino propose extending the flow analysis for constructors to ensure each constructor assigns to each of the fields. But this restriction does not suffice because the constructor can use the object it is constructing *before* assigning to all the fields. (This problem is precisely why Java initializes object fields implicitly.)

Therefore, they further distinguish the types of objects whose constructors have not completed. For non-`NULL` fields of objects of such types, the value may be `NULL`, but only non-`NULL` values may be assigned. After an assignment, the

flow analysis may assume the value is not `NULL`. In terms of the formalism in this chapter, this technique essentially adds a new abstract rvalue ALL*' and allows $\Gamma \vdash_{\text{wf}} \tau@, \text{ESC}, \text{ALL}^{*\prime}$. If $\Gamma(x) = \tau@, \text{ESC}, \text{ALL}^{*\prime}$, then we cannot dereference x, but we can assign a pointer to it. In the analysis, such an assignment makes $\Gamma(x) = \tau@, \text{ESC}, \text{ALL}@$. Type preservation holds because $\Gamma \vdash \text{ALL}@ \leq \text{ALL}^{*\prime}$.

For arrays of non-`NULL`, Fähndrich and Leino require a run-time check that the program has assigned to every array element. Cyclone is even more restrictive because it requires immediate initialization of arrays.

Another language-based approach, more popular among functional languages, is to eliminate `NULL` and require programmers to use discriminated unions. Retrieving an actual pointer from a possible pointer requires special syntax, such as pattern-matching. One drawback is that actual pointers are not implicit subtypes of possible pointers.

Under-Specified Evaluation Order Given the number of languages that have under-specified evaluation order, there has been surprisingly little work on source-level flow analysis for such languages.

For example, Scheme [179] has a "permutation semantics" for function application, in the sense described in Section 6.2. The Scheme community has extensively researched approaches to control-flow analysis, i.e, statically approximating the functions to which an expression might evaluate [11, 124, 183]. To my knowledge, all such presentations have assumed a fixed evaluation order. (Some analyses are flow-insensitive, in which case evaluation order is irrelevant.) This assumption is reasonable when the purpose of the analysis is optimization because a compiler can perform the analysis after choosing an evaluation order.

For "actual C semantics," just the language definition has been a large source of confusion. The ISO standards committee has considered several complicated formalisms of what sequence points are and what is allowed between them [68, 147, 176]. Using another reasonable formalism, Norrish used a theorem prover to show that most legal expressions are actually deterministic—evaluation order cannot affect the result [168, 167]. In fact, they are all deterministic, but Norrish's formal proof does not prove this result for expressions with sequence points in under-specified evaluation-order positions, such as `(x,y)+(z=3)`. Interestingly, the unsoundness of the "join approach" in this chapter results from the same class of expressions, but this fact may be coincidence.

Norrish's result provides an important excuse for flow analyses examining C code: If we *assume* that the source program is legal, then the analysis can soundly choose any evaluation order for expressions. If this assumption does not hold, the program is undefined, so analyzing it is impossible anyway. This excuse does not

work for "C ordering semantics."

Splint [189] attempts to find expressions that are undefined because of evaluation order, but this analysis is incomplete. Its other analyses assume a left-to-right evaluation order.

CCured [164, 38] compiles C code in such a way as to ensure safety. It implements C with left-to-right evaluation, which is certainly compatible with the C standard.

Incorporating Points-To Information Cyclone's analysis incorporates simple must points-to information. The primary motivation is to support delayed initialization of heap-allocated memory, i.e., `malloc`. Many compilers do not include points-to information in other flow analyses. Instead, they precompute points-to information with a different analysis. This analysis can provide a set of possible abstract locations to which each expression might evaluate. Subsequent analyses can then use these sets to approximate the effect of assignments and whether data might be uninitialized or `NULL`. Because Cyclone's use of points-to information is rather unambitious and any analysis must be part of the language definition, having one analysis is a good choice. Steensgaard [190] presents a particularly fast flow-insensitive interprocedural points-to analysis. This work also describes slower flow-sensitive approaches. More recent work has refined and extended Steensgaard's basic approach. Andersen's dissertation [6] develops a points-to analysis for C so that his partial evaluator can avoid overly pessimistic assumptions about pointers.

Other program analyses reason about pointers and can determine that two pointers are the same. A particularly powerful approach is *shape analysis*, in which shape graphs statically approximate the structure of a run-time heap. In earlier work [128, 42], nodes in the graph correspond to allocation sites in the source program, somewhat like Cyclone's flow analysis uses allocation sites to define the space of abstract rvalues. This approach makes it difficult for the analysis to prove properties about data structures of unbounded size. Indeed, there is no way in Cyclone to create a list of uninitialized data unless each list element is allocated at a different program point. The more sophisticated shape analyses of Sagiv, Reps, and Wilhelm [181] eschew a correspondence between shape-graph nodes and allocation sites. The shape graphs also have a notion of unsharedness that corresponds to linearity in type-theoretic terminology and allows the graphs to summarize the structure of some data structures of unbounded size.

Dor, Rodeh, and Sagiv have used shape analysis and pointer analysis to find errors in C programs that manipulate pointers [58, 59]. Their approach is conservative: If it reports no errors, then the program cannot leak memory, access

deallocated storage, or attempt to dereference `NULL`. In a very rough sense, the main difference between this work and Cyclone's analysis is the technique for generating the pointer information. Their shape analysis is much more sophisticated, which leads to the usual advantages and disadvantages with respect to performance, accuracy, understandability, etc.

Type-Theoretic Approach Smith, Walker, and Morrisett's work on alias types [186] develops a type system with points-to information roughly comparable to Cyclone's analysis. Several differences deserve explanation. First, they distinguish type-level location names from term-level locations whereas Cyclone uses variables (or allocation sites) for both purposes. As a result, Cyclone rejects this code:

```
int **z;
if(e) z = malloc(sizeof(int*));
else  z = malloc(sizeof(int*));
*z = new 17;
```

In the alias-types framework, the conditional's continuation (the final assignment) would be polymorphic over a location name and the conditional's branches could both jump to the continuation by instantiating this type-level name differently.

Second, their term language is an idealized assembly language with control transfers that amount to continuation-passing style. As a result, locations and location names never leave scope, so their system does not encounter the complications that led us to abandon α-conversion in this chapter's formalism. Relatedly, sequences $s_1; s_2$ in their language restrict s_1 to primitive instructions such as assignment or memory allocation, and primitive instructions must precede some s_2. This restriction avoids the need for typing judgments to produce typing contexts.

Third, locations escape via an explicit type-level application of an unescaped (linear) location name to a polymorphic function expecting an escaped (nonlinear) location name. This technique replaces the escapedness and abstract-ordering judgments in our formalism.

Fourth, they allow explicit deallocation of unescaped locations. It should be straightforward to add a `free` primitive to Cyclone that takes a pointer to an unescaped location and forbids subsequent use of the location.

Like Cyclone, the alias types work allows run-time tests to refine flow information, such as whether a possibly-`NULL` is actually `NULL`, but only if the tested location is unescaped. Prior to the work on alias types, Typed Assembly Language [157] could not support cyclic lists as presented in Section 6.3.

Subsequent work [212] combined location names with recursive types to express aliasing relationships in data structures of unbounded size. This extension subsumes linear type systems [206, 202], which can express only that a pointer refers

to a location to which no other pointer refers. Like shape analysis, this technology could allow us to allocate a list of uninitialized data and then initialize each element of the list.

Instead of formalizing Cyclone's analysis as a type system, we could use abstract interpretation [49]. In theory, abstract interpretation and type systems are both sufficiently powerful foundations for program analysis, but the different formalisms have different proof-engineering benefits.

The type-safety proof in Appendix D shows that the declarative formulation of the flow analysis is strong enough to keep the dynamic semantics from getting stuck. By examining the dynamic semantics, we see that it is impossible to dereference 0 or uninitialized values, so the correctness of the analysis follows as a metalevel corollary. It took considerable effort to revise the analysis to produce an algorithm of similar power, and we did not prove any notion of similarity.

To contrast, an abstract-interpretation approach to the problem would define an abstract semantics (such as how expressions manipulate abstract values) much like our type system. But instead of using syntactic techniques to prove safety, we would prove that the abstract semantics and the dynamic semantics are appropriately related by an abstraction function that maps concrete values to abstract values [49, 166]. Having established that the abstract semantics was a valid abstract interpretation, we would be guaranteed that it was correct, in the sense that an expression that abstractly evaluates to some r must concretely evaluate to some v such that v has abstract value r. As a result, the dynamic semantics cannot get stuck. Furthermore, because the abstract domain does not have infinite chains of the form $r_1 \leq r_2, \ldots$ where each r_i is distinct, we know an algorithm can implement the abstract interpretation.

With our type system, proving type preservation did not require changing the term syntax, but executing loop bodies did require systematic renaming in the dynamic semantics. Abstract interpretation could allow implicit α-conversion of term variables, but proving that it was a valid abstract interpretation would require maintaining some connection between different copies of a loop body. Otherwise, we cannot prove that an expression at some "program point" always evaluates to a value with certain properties. A standard approach is to change the term syntax to include labels (which do not α-convert) on all terms.

This dissertation does not determine which approach is "better." It does show that type systems can describe flow-sensitive compile-time information including pointer analysis. Furthermore, the syntactic approach to soundness that Wright and Felleisen advocate [219] can establish safety.

Chapter 7

Array Bounds and Discriminated Unions

In preceding chapters, we developed an advanced type system and flow analysis for preventing several flavors of safety violations in C code. In this chapter, we sketch how to use similar techniques for preventing incorrect array indexing and misuse of union values. Array-bounds violations violate memory safety directly: without restricting the value of e, the expression `arr[`e`]=123` can write `123` almost anywhere. Misusing union values also leads to unsafe programs: Writing to a union through one member and then reading through another member is equivalent to an unchecked type cast.

For both problems, we shall make the simplifying assumption that some *(unsigned) integer* determines the correct use of an array (by representing its length) or union (by indicating the member most recently written to). This integer could be known at compile-time, or it could be stored and tested at run-time. Most implementations of high-level languages simply store array lengths and discriminated-union tags with the corresponding data objects. Accessing the data objects involves implicit checks against these integers. In Cyclone, it is more appropriate to expose the checks and data-representation decisions to programmers.

Extending the techniques from earlier chapters is promising. By introducing compile-time integers and type variables that stand for them, we can use quantified types and type constructors to encode the connection between integers and the objects they describe. We can also extend our flow analysis to approximate the value of mutable integers (so we can use them for array indexing) and the current type of value in a union.

This chapter sketches the extensions informally and evaluates them. We do not present a formal semantics or a type-safety result. Moreover, these features are quite experimental in the actual Cyclone implementation. Current applications

make little use of them. Nonetheless, the extensions seem natural, sound, and true to the approach taken for other problems. However, the material in this chapter cannot support pointer arithmetic.

We also do not consider in detail how to decide nontrivial arithmetic facts at compile-time. The specific form of arithmetic constraints and a decision procedure over them should be orthogonal to the basic approach developed in this chapter. We briefly describe an "interval approach" more like the abstract rvalues in the previous chapter and a "set of constraints" approach closer to the current Cyclone implementation. Choosing a powerful and usable arithmetic remains ongoing work.

The rest of this chapter is organized as follows. Section 7.1 describes the extensions to the type system for describing the lengths of arrays. Section 7.2 uses these types and an extended flow analysis to enforce safe array-indexing. We delay further discussion of `union` types until Section 7.3. This section presents more sophisticated `union` types than C has and further extends the flow analysis to reason about union values. Section 7.4 evaluates our extensions. Section 7.5 discusses related work.

Throughout this chapter, we use `uint_t` as an abbreviation for `unsigned int`.

7.1 Compile-Time Integers

This section explain how we add known and unknown integers to the type system to reason about array lengths. We first add tag types (often called singleton-integer types in the literature) and modify pointer types to include lengths. These additions suffice for type-checking the Cyclone constructs for creating arrays. We then present examples using quantification over compile-time integers. Finally, we describe subtyping induced by our type-system changes.

7.1.1 Types

Just as Chapter 4 introduced a kind `R` for region names and Chapter 5 introduced a kind `L` for lock names, we introduce a kind `I` for integer types. We then add types for each positive integer. For example, the *type* 37 has kind `I`. The *term* 37 does not have type 37 because all terms have kind `A`.

We can give the term 37 the type `tag_t` <37>. That is, `tag_t` is a type constructor that takes a type kind `I` and produces a type of kind `A` (in fact, `B`). This constructor is analogous to the constructor `region_t` in Chapter 4, which produced the type for a handle given a region name. As expected, pointer types include compile-time integers for their length. For example, `int*{37}` describes pointers to arrays of 37 integers. (The braces in the type are just syntax because

`int*37` looks odd.) In general, we build a pointer type from an element type and a type of kind `I`. An omitted length is short-hand for `{1}`.

These additional types suffice for type-checking the constructs that create arrays and pointers to them. As in C, we can give a variable an array type provided that the array size is known. For example, `int x[23];` declares `x` to hold an array of length 23. When `x` is used in C, its type is implicitly promoted to `int*`. In Cyclone, we also have implicit promotion, but the resulting type is `int*{23}`.

To build an array with a length that depends on run-time information, C requires `malloc` (or `salloc`, which we do not consider). Chapter 6 described why Cyclone cannot determine that arrays created with `malloc` get initialized. Therefore, we use special syntax for creating and initializing an array: The initializer form `{for x<`e_1 `:` e_2`}` creates an array of length e_1 where element i is initialized to e_2 (with i substituted for `x` in e_2).

So `new {for x<`e_1 `:` e_2`}` produces a pointer to a heap-allocated array. The type of such an expression is τ`*{`i`}` where e_2 has type τ (assuming `x` has type `uint_t`) and e_1 has type `tag_t <`i`>`. For example, `new {for x<23 : x*x}` has type `uint_t*{23}` and points to an array of squares.

But with the type system presented so far, the typing rules are still too restrictive to create an array whose length depends on run-time information. To do so, we add type variables of kind `I`, i.e., unknown compile-time integers. For example, a variable `y` could have type `tag_t <`α`>` and `new {for x<y :` e_2`}` would have a type of the form τ`*{`α`}`. As a convention, we will use i and j to range over type variables of kind `I`. We now turn to introducing and using such type variables.

7.1.2 Quantified Types

As expected, universal quantification lets functions take unknown integer constants. For example, this function makes an array with a length the caller specifies:

α`*{`i`} f(tag_t<`i`> len,` α `val) { return new {for x<len : val}; }`

Clearly, the callee does not know the length. But we need one more extension for a caller to compute a length at run-time. The key is to realize that `uint_t` is equivalent to $\exists i.$`tag_t <`i`>`; an integer value is some unknown constant. Hence we can use an existential unpack to convert from `uint_t` to a tag type. For example, this code truly creates an array of unknown length:

```
let len<i> = fgetc(stdin); // read from input and unpack
int*{i} arr = new {for x<len : 0};
```

The first declaration is peculiar syntax for unpacking a `uint_t`. As usual, an unpack introduces a type variable (here with kind `I`) that is in scope for the rest

of the block. It is important that the unpack uses a fresh location for the value of type `tag_t` <i>. Otherwise, we can violate type safety as explained in Section 3.3.

Existential quantification is also important for user-defined types. A simple example lets programmers store array bounds with a pointer to the array:

```
struct Arr<α> { <i>
  tag_t<i>  len;
  α*{i} arr;
};
```

Letting users specify where bounds are is more flexible than the compiler inserting implicit bounds for each array. For example, we could define a type where the same tag describes two arrays:

```
struct TwoArr { <i>
  tag_t<i>    len;
  int*{i}    arr1;
  double*{i} arr2;
};
```

However, this flexibility is limited. We cannot indicate that one array is 3 elements longer than another array of unknown size unless we add types of the form $i+3$. In other words, our symbolic arithmetic at the type level does not have operators like addition. We also still require all array elements to have the same type. This restriction precludes an example where element i of an array points to an array of length i. (Such a data structure could represent a triangular matrix.)

We can support types of unknown size to a limited extent, as in this example:

```
struct FlatArr { <i>
  tag_t<i> len;
  int arr[i];
};
```

C does not allow such types, but unchecked casts let programmers work around the limitation. In Cyclone, it suffices to give kind `A` to type `struct FlatArr` because the size of an object with this type is unknown. Programmers cannot create arrays or have local variables of such types. We also disallow fields after `arr`.

7.1.3 Subtyping and Constraints

Our type-system extensions lead to two natural notions of subtyping. First, `tag_t` <τ> is a subtype of `uint_t`. Second, we can treat a longer array as a shorter

array. For example, `int*{37}` is a subtype of `int*{36}`. For known compile-time integers, deciding subtyping is obvious. For unknown compile-time integers, we can use subsumption only if we know certain compile-time inequalities. For example, we can subsume `int*{`i`}` to `int*{`j`}` if we know $j \leq i$.

To track such inequalities, we can use constraints much like we did in Chapters 4 and 5. Quantified types can introduce constraints of the form $\tau < \tau'$ (or $\tau \leq \tau'$). For example, this function `f` can access the first 37 elements of the array it is passed, but the caller can still know that the array returned is longer:

`int*{`i`} f(int*{`i`} :` $37 \leq i$`);`

Hence, one way to introduce a constraint is for the caller to satisfy the inequality at compile-time, such as by passing a pointer of type `int*{40}`. Another way is to use run-time tests. For example, if e_1 has type `tag_t <`τ_1`>` and e_2 has type `tag_t <`τ_2`>`, then `if(`e_1`<`e_2`)` s_1 `else` s_2 lets us assume $\tau_1 < \tau_2$ in s_1 and $\tau_2 \leq \tau_1$ in s_2. Hence, the set of assumed constraints depends on the program point.

More sophisticated constraints lead to richer notions of arithmetic. For example, the current Cyclone implementation accepts $i - j < 37$ as a constraint. Important but conflicting design goals for a constraint language are expressiveness and having a tractable decision procedure for determining if a set of constraints implies another set of constraints. This procedure is necessary for eliminating quantified types and—as discussed below—for array indexing.

7.2 Using Arrays

Having extended the type system to account for the correspondence between array lengths and integers holding those lengths, we now turn to ensuring that array-subscript operations (`(`e_1`[`e_2`]`) stay within bounds.

A simple solution would replace C's subscript operation with a ternary operator `sub(`e_1`,`e_2`,`e_3`)`. This operator would be equivalent to `(`e_1`[`e_2`]` if e_2 evaluated to less than e_3, else it would raise an exception. It suffices to treat `sub` as a polymorphic function taking arguments with types α`*{`i`}`, `uint_t`, and `tag_t <`i`>`, respectively.

This solution gives no way for programmers to ensure statically that subscript operations do not fail, nor does it let programmers remove provably unnecessary checks. We noticed the same disadvantages for implicit `NULL` checks when dereferencing possibly-`NULL` pointers in Chapter 6. We consider how to extend that chapter's flow-sensitive analysis to reason about values used to index arrays.

One approach uses a *conjunction of intervals* to approximate the value of an expression with type `uint_t`. In the terminology of Chapter 6, we extend abstract rvalues with conjunctions of intervals. We can always approximate a `uint_t` with

$[0, 2^{32} - 1]$ (assuming a 32-bit machine). To be sound in the presence of aliasing, we use this abstract rvalue for the contents of any escaped location.

For unescaped locations, we can often be more precise. For example, when we subsume an expression of type `tag_t` $<\tau>$ to `uint_t`, the resulting expression can have abstract rvalue $[\tau, \tau]$. That is, we still know its value. The result can then flow to other expressions. For example, because 37 has type `tag_t` $<37>$, the declaration `uint_t x = 37;` gives `x` the abstract rvalue $[37, 37]$.

We can also use run-time tests to produce more precise intervals. For example, consider `if(x<`e`)` s_1 `else` s_2 where `x` is unescaped and e has abstract rvalue $[i, i]$. We can type-check s_1 knowing `x` holds a value in the range $[0, i-1]$ and s_2 knowing `x` holds a value in the range $[i, 2^{32} - 1]$. Allowing conjunctions is important because we may not know the relative ordering of unknown compile-time integers. For example, if successive run-time tests ensure `x` is less than i and less than j, we have no reason to prefer the interval $[0, i - 1]$ over $[0, j - 1]$ or vice-versa.

The obvious purpose of these abstract rvalues is to check subscript operations. If e_1 has type τ_1`*{`τ_2`}`, then we allow e_1`[`e_2`]` only if the abstract rvalue for e_2 implies e_2 is less than τ_2. We do not consider the details of this decision procedure.

With run-time tests, the flow analysis could influence type-checking by introducing compile-time inequalities. Unfortunately, the current Cyclone implementation strictly phases type-checking (including constraints) before flow analysis.

The current Cyclone implementation takes an approach less similar to the previous chapter: It extends *constraints with mutable integers*. This approach can track inequalities such as $x < y$ even without other information about x and y. However, it is more difficult to compute accurate approximations as necessary. For example, assume we know $x < y$ and $y < z$. Then if y escapes or we join with an abstract state with $x < z$, we should still conclude $x < z$. Although we should be able to achieve this expressiveness, the current implementation does not.

Nonetheless, both approaches accept some important idioms. First, programmers can write functions like `sub` when they wish to rely on dynamic checks:

```
τ sub(τ*{i} arr, uint_t elt, tag_t<i> len) {
  if(elt < len)
    return arr[elt];
  throw ArrayException;
}
```

Second, we can verify straightforward loops such as in this function that adds the elements in an array:

```
int add(int*{i} arr, tag_t<i> len) {
  int ans = 0;
  for(int x=0; x<len; ++x)
    ans += arr[x];
  return ans;
}
```

We can also accept some more sophisticated examples. For example, this function returns an array that can hold twice as many elements:

```
struct Arr<α> { <i>
  tag_t<i>  len;
  α*{i} arr;
};
struct Arr<α> double(struct Arr<α> s) {
  let Arr{<i> .len=len, .arr=arr} = s;
  if(len==0)
    return s;
  let newlen<j> = len*2;
  α*{j} newarr = new {for x < newlen: (x < len ? arr[x] : arr[0])};
  return Arr{.len=newlen .arr=newarr};
}
```

The existential type hides the correspondence between the lengths of the argument and result. The initializer for `newlen` type-checks because we can subsume the type of `len` from `tag_t<i>` to `uint_t`. Run-time tests let the flow analysis accept the array-subscript expressions in the initializer of `newarr`. First, we execute `arr[x]` only if `x<len`. Second, the earlier test `len==0` ensures $0 < i$.

Note that we did not assume `len*2>len`. The analysis does not perform this level of mathematical reasoning, nor is it sound if `len` may be greater than 2GB (on a 32-bit machine). In general, arithmetic overflow makes soundness more difficult.

This safe example is beyond what the current implementation can support:

```
int acmp(int f(α,β), α*{i} a1, β*{j} a2, tag_t<i> l1, tag_t<j> l2) {
  uint_t minlen = (l1 < l2) ? l1 : l2;
  for(uint_t x=0; x < minlen; ++x) {
    int ans = f(a1[x], a2[x]);
    if(ans != 0)
      return ans;
  }
  return l1 - l2;
}
```

To accept this function, the abstract state after the initialization of `minlen` must imply `minlen`$\leq$`l1` and `minlen`$\leq$`l2`. As mentioned above, a join operation with this accuracy should be possible.

It is also possible with the conjunction-of-intervals approach; the key to the join operation is to expand the two abstract states to include intervals that are redundant only due to the compile-time inequalities. In our example, we add $[i, j]$ to the intervals for `minlen` in the true-branch (because $i < j$ and we have the interval $[i, i]$). Similarly, we add $[j, i]$ to the intervals for `minlen` in the false-branch. Second, we produce a joined state assuming only the compile-time inequalities for the program-point after the join. In our example, there are no such inequalities. However, given $[i, j]$ from the true-branch and $[j, j]$ from the false branch, we can conclude $[0, j]$. Analogously, given $[i, i]$ from the true-branch and $[j, i]$ from the false-branch, we can conclude $[0, i]$.

7.3 Using Discriminated Unions

This section explains how we can use compile-time integers to enforce the safe use of unions. The key addition is to enrich `union` types such that each member has an associated constraint. For example, we can use these declarations to encode some arithmetic expressions:

```
struct Exp;
struct TwoExp {
  struct Exp * exp1;
  struct Exp * exp2;
};
union U<i> {
  int           num;        @requires i==1;
  struct Exp *  negation;   @requires i==2;
  struct Exp *  reciprocal; @requires i==3;
  struct TwoExp plus;       @requires i==4;
  struct TwoExp minus;      @requires i==5;
  struct TwoExp times;      @requires i==6;
  struct TwoExp divide;     @requires i==7;
};
struct Exp { <i> tag_t<i> tag; union U<i> u; };
```

For now, suppose the type `union U<4>` means only the `plus` member of the value is accessible (for reading or writing). We can then use the existential quantification in the definition of `struct Exp` to abstract which member is accessible. Clients

can unpack such values and test the value that was in the `tag` field to regain the information necessary to access the value in the `u` field. Clients can also mutate `struct Exp` values to hold different variants so long as both fields are mutated simultaneously.

In C, we could use a similar encoding, but nothing in the type system would represent the connection between the `tag` field and the member of the `u` field that was written. Therefore, the type-checker cannot check that code checks the `tag` field and reads only through the appropriate member of `u`.

The different members of a `union` type must be "guarded" by requires clauses that a decision procedure can prove do not overlap (i.e., no two can both hold). Given a value of type `union U<`τ`>`, we allow accessing member `f` only if the guard for `f` holds. Compile-time inequalities can produce such information. Continuing our example, we can allow code like the following:

```
let Exp{<i> .tag=t, .u=u} = e;
switch (t) {
case 1: /* use u.num */ break;
case 2: /* use u.negation */ break;
default: break;
}
```

In the first branch of the switch statement, we know `u` has type `union U<`i`>` and i equals 0. The other branches are similar.

Our rules for compile-time inequalities are expressive enough to accept code that uses binary search to determine tag values. However, we still require determining the exact variant before allowing access, even if safety does not demand it. For example, if $i > 4$, it is safe to cast from `union U<`i`>` to `union U<6>`.

Using existential types (as in `struct Exp`) and type constructors (as in `union U`) in this way lets us encode "discriminated unions" (where the tag is present at run-time). As desired, we let programmers choose where to put the tag and how to test its value.

However, these techniques rely on type invariance: So far, the only way to change which member of a `union` value is accessible is to mutate an existential package that contains it. For escaped locations, we do not endeavor to do better. There are unknown pointers to the location. To ensure that they access the correct member after the mutation, we require either that the correct member does not change (e.g., if the location has type `union U<3>`) or that a tag is updated at the same time (by using an existential type).

For unescaped locations, we can use the flow analysis to allow changes to which member is accessible. If `x` is an unescaped location of type `union U<`τ`>`, then the

flow analysis tracks the last member that has been written to. Specifically, suppose the definition of `union U` has members $f_1, \ldots, f_n$, and r ranges over abstract rvalues. Then the possible abstract values for `x` are NONE (the location is possibly uninitialized) and $f_i(r)$ (the last member written was f_i and it contains a value that r approximates).

Assuming `x` is unescaped, we allow the right-expression `x.f` only if the flow analysis determines that `f` is the last member written to. However, as a left-expression, we allow any `x.f`. The result of assigning through a member can change the abstract rvalue for `x`. To join two control-flow paths where the last members written two are different, we can forget that `x` is initialized.

Essentially, we let unescaped locations "change type" (e.g., from `union U<3>` to `union U<2>`). Nonetheless, at any point when the location escapes (i.e., any program point where not all pointers to the location are known exactly), the last member written to must be the one indicated by the location's declared type.

This flexibility lets us reuse unescaped locations, as in this example:

```
void f(struct Exp* e1, struct Exp* e2) {
  union U<7> u; /* exact type irrelevant in this example */
  struct Exp e;
  if(flip()) {
    u.num = 42;
    e = Exp{.tag=1, .u=u};
  } else {
    u.plus = TwoExp{.exp1=e1, .exp2=e2};
    e = Exp{.tag=4, .u=u};
  }
}
```

To check this example, the type-checker should record implicit casts from `union U<7>` to `union U<1>` and `union U<4>`. The flow analysis can ensure these casts are safe where they occur because `u` is unescaped. These implicit casts may not interact well with type inference because they give the type-checker an awkward flexibility.

Another design choice is to distinguish union types that can change members from those that cannot. Then we would never allow the latter to escape. Instead, it suffices to have one style of union type that has different restrictions depending on a location's escapedness.

7.4 Evaluation

We have described how to ensure safe use of arrays and unions in Cyclone. The key addition to the type system was compile-time integers, including type variables standing for unknown constants. For the flow analysis, we extended our abstract states to integer constraints and the accessible member of union values.

Compared to some problems in earlier chapters, two factors make arrays and unions more difficult (and the solutions more complicated):

1. Programs often manipulate integers in ways that are safe only because of nontrivial mathematical facts. In other words, numbers enjoy much more interesting relations than type, locks, initialization state, etc.

2. Implementing run-time checks for `NULL` pointers is straightforward because the check needs only the possible-pointer. Run-time checks for array lengths and union members require an appropriate tag. In C, this tag is not passed to the necessary operators.

The techniques we developed mostly address the latter point by using the type system to connect tags to data and the flow analysis to separate run-time tests from data access. Incorporating arithmetic in the presence of mutation and overflow remains ongoing work. Fortunately, choosing a constraint language and decision procedure appears largely orthogonal.

Another important aspect of our design is that it makes it explicit that data objects like discriminated unions and arrays carrying their lengths are existential types. Therefore, we can use the technology in Chapter 3 to ensure we use them safely.

We now describe several specific limitations that our approach suffers. We then consider two advanced idioms discussed earlier in the dissertation.

The most basic assumption we make is that the tag describing an array length or union value is either known statically or held in a particular location at run-time. However, safe C programs may have other ways to determine a value's tag. One example is storing an array's length divided by 17 instead of the length. Another example was the representation of a triangular matrix we described earlier. A far more common example is a nul-terminated string: In C, the convention is that it is safe to access successive string elements until encountering a 0. This convention is a completely different way of determining an array length at run-time. Cyclone has some experimental support for nul-terminated strings, which we do not discuss here. A final example is programs that go through "phases" in which all union values of some type use one member and then they all use another member.

Another major limitation is the lack of support for pointer arithmetic. For example, we allow code like `while(++i < len) f(arr[i]);`, but not code like `while(++arr < end) f(*arr);`. On some architectures, C compilers can produce much faster code for the latter. The actual Cyclone implementation allows pointer arithmetic only for pointers that have implicit bounds fields and run-time checks. That is, for a given pointer, programmers can control data representation or use (relatively slow) pointer arithmetic, but not both.

A common use of `union` types allows convenient access to overlapping subranges of bits (that are not pointers). For example, if a value has several small bit fields with total size less than `sizeof(int)`, we can have one `union` member with a `struct` type suitable for reading fields and another `union` member with type `int`. The latter makes it easy to set all fields to 0 simultaneously, for example. Technically, C forbids reading through one member if another member was last written, but conventional implementations allow such idioms. Assuming a conventional implementation, it is safe for Cyclone to allow reading through a `union` member with a nonpointer type. The Cyclone implementation allows such access.

A final point is that prototypes of the form `void f(int n,` τ `arr[n]);` are syntactically more pleasant than `void f(tag_t<`‘`n> n,` τ `arr*{`‘`n});`. The latter makes an important distinction: mutating `n` does not change the length of `arr`. Nonetheless, allowing the former as syntactic sugar is straightforward.

We now consider two C idioms we encountered in earlier chapters, before we had support for compile-time integers. The first is a generic function that makes a copy of data. For example, the C library provides a function with this prototype:

```
void* memcpy(void *out, const void* in, size_t n);
```

The most similar prototype we can give in Cyclone is this one:

```
α*{i} memcpy(α*{i} out, const α*{j} in, sizeof_t<α> s, tag_t<j> n
             : i ≥ j);
```

The Cyclone version suffers from several problems. First, it is not implementable in Cyclone because there is no way to copy a value of unknown size. However, we can give this type to a function implemented in C. Second, we had to represent the amount of data to copy with two arguments, the size of the element type (α) and the number of elements (j). We could overcome this limitation by enriching the language of compile-time arithmetic expressions enough to write `tag_t<`j`*sizeof_t<`α`>>`. Third, it does not prevent the caller from passing overlapping memory regions (i.e., arguments where `out+s*n > in` or `in+s*n > out`). In C, `memcpy` is undefined if the regions overlap. However, the similar `memmove` function allows overlap. The trade-off is that `memcpy` may execute faster.

Our second example proves even less successful. In Chapter 6 we described an "initializes" attribute for function parameters. This attribute indicates that the caller should pass a non-`NULL` pointer to an unescaped and possibly uninitialized location. The callee must initialize this location before returning. This *ad hoc* extension still does not allow the callee to return a value indicating whether it initialized the location. Given the technology developed in this chapter, we would hope to use a `union` type and a tag type to encode this idiom. Here is a possible first step, exploiting that actual Cyclone does not require initializing nonpointers:

```
union U<i> {
  int x;  @requires i==0;
  int *p; @requires i==1;
};
tag_t<i> f(union U<i> @u) attribute(initializes(1)) {
   if(e) {
     *u = new 0;
     return 1;
   }
   return 0;
}
```

Unfortunately, universal quantification is incorrect for this function. It is the callee that chooses the tag, not the caller. The correct quantification is existential: there exists some integer i such that the callee returns the value of type `tag_t<i>` and initializes `*u` through the appropriate union member. So at minimum, we need to extend Cyclone with existential quantification over function types.

Moreover, the caller needs some way to unpack the existential that the function call introduces. But no data object contains the function's result and the location the callee referred to as `*u`. Put another way, if the caller passes `f` some `&x`, the type of `x` after the callee must be bound by the same existential as the function result. To do so seems to require some special syntax for packaging the function result with `x`. It is much simpler to abandon the initializes attribute and rewrite `f` to return an existential type holding the tag and the union value:

```
struct Pr { <i> tag_t<i> t; union U u; };
struct Pr f() {
  if(e)
    return Pr{.t=1, .u=new 0};
  return Pr{.t=0, .u=0};
}
```

7.5 Related Work

This section discusses some other projects that prevent unsafe array or union accesses, or reason about integer values at compile-time. Far too much work exists for a thorough review. We therefore focus on systems for preventing array-bounds violations in C, static analyses for reasoning about integer values, and languages that express array lengths and union tags in their type systems. Considerable overlap in the first two areas makes the distinction somewhat arbitrary.

7.5.1 Making C Arrays Safe

The simplest way to prevent array-bounds violations in C code is to compile C such that all pointers carry the size of the pointed-to object at run-time. Run-time checks can terminate a program as soon as a violation occurs. Obviously, this approach loses static assurances and changes the data representation C programmers expect (but are not promised).

The first project I am aware of that uses this technique as part of a C implementation that ensures safety is Safe-C [12]. In Safe-C, pointers also carry information to determine whether the pointed-to object has been deallocated. One problem with changing data representation is that it requires recompiling the whole program, which is impossible if some source code (e.g., for the standard library) is unavailable. Other work [129] avoids this shortcoming by storing the auxiliary information in a table indexed by machine addresses. Of course, a pointer dereference must now look up the information in the auxiliary table.

These systems suffer substantial performance degradation because pointers always occupy extra space and every pointer dereference requires run-time checks. The CCured project [164, 38] uses a whole-program static analysis to avoid most of this overhead. This analysis can avoid changing data representation when a pointer need only point to an "array" of length 1. It can also use only an upper bound when negative index expressions and pointer subtraction are not used. The whole-program static analysis is linear in the size of the program. Programmers must specify the representation for pointers that are passed to or returned from code not compiled by CCured. It does not appear that CCured can exploit that a user variable already holds an array-length. The project has focused on arrays; uses of unions are treated as casts with run-time checks. Finally, CCured provides special support for nul-terminated strings by making an implicit terminator inaccessible to user programs. Chapter 8 compares Cyclone and CCured in general. The published work on CCured [164] provides an excellent description of some commercial tools with similar goals.

Other projects have focused on the misuse of strings and buffers that hold them.

For example, Wagner et al. [208, 207] automatically found several buffer overruns in real code that had already been audited manually. They use integer intervals as a primary abstraction and approximate each integer variable with an interval. They generate interval constraints completely before solving the constraints. For scalability, the constraint generation is flow-insensitive. They model character buffers with the length of the string they hold (where the first nul-terminator is) and the allocated size. (Recall Cyclone, as presented in this chapter, does not reason about nul-terminators.) The analysis knows how important library routines, such as `strncpy` and `strlen` affect and determine abstract buffer values. As a bug-finding tool, their work is unsound with respect to aliasing. The language for integer constraints is more sophisticated than in Cyclone because it allows operations like addition. However, its constraint solver is based on "bounding boxes," which are more approximate than most approaches described in Section 7.5.2.

Dor et al. [60] use a more precise analysis that can find some subtle safety violations without generating many false positives. It also relies on integer analysis, but it uses polyhedra that are more precise than bounding boxes. The analysis is sound (the absence of errors guarantees the absence of bound errors), but functions require explicit preconditions and postconditions. Moreover, the analysis does not handle multilevel pointers and has not been applied to large programs.

7.5.2 Static Analysis

This section describes more general approaches to static reasoning about integer values, array lengths, and union values. Compared to the work described above, these projects have less essential connection to C.

One approach to forbidding array-bounds errors is to generate a *verification condition* that implies their absence. Given $e_1[e_2]$, the verification condition would require a precondition for this expression that implied e_2 evaluated to a value less than the length of the array to which e_1 evaluates. A theorem prover can try to prove the verification condition. If the verification-condition generator and theorem prover are sound, then such a proof establishes the absence of bounds errors. This architecture underlies extended static checking, as in ESC/Java [76], and proof-carrying code, as in the Touchstone certifying compiler [161, 162]. It separates the problems of finding a mathematical fact that must hold and determining that the fact does hold. However, theorem provers are incomplete and can be slow.

Other projects have investigated more traditional compiler-based approaches to bounds-check elimination. For example, Gupta [101] describes a straightforward approach to using flow analysis for reducing the number of bounds checks. The analysis is more sophisticated than in Cyclone for at least two reasons. First, it interprets arithmetic operators, including multiplication and division. Second,

it determines when it is safe to hoist bounds-checks out of loops. Cyclone is less interested in the latter because it should suffice for programmers to hoist checks themselves and have the analysis verify that the result is safe. More recent work by Bodik, Gupta, and Sarkar [25] eliminates bounds-checks using a *demand-driven* analysis (given a check to consider for elimination, it attempts to avoid work irrelevant to that check) over a *sparse* representation (it does not operate over a full control-flow graph). Their aim is to support simple, fast bounds-check elimination. This work also describes a wide variety of previous approaches to bounds-check elimination.

Rugina and Rinard [180] use a symbolic analysis to approximate the values of pointers, array indices, and accessed-memory regions. By producing a constraint system that can be reduced to a linear program, they can avoid many limitations of fixpoint-based flow analyses. One application of approximating the memory that an expression might access is the static detection of array-bounds errors.

Flow-based analyses invariably need to compute the implications of integer constraints involving unknown integers. The literature includes some well-understood solution procedures for restricted classes of inequalities (e.g., linear inequalities). The Omega Calculator [175] is a popular tool that simplifies all Presburger formulas (which can contain affine constraints, logical connectives, and universal and existential quantifiers). Such formulas are intractable in theory, but the calculator has proved efficient in practice.

Some of the work on bounds-check elimination described here claims that simple arithmetics (though more sophisticated than what Cyclone supports) suffice. In contrast, the data-dependence community uses somewhat similar techniques to optimize numerical applications. Rather than detect bounds violations, they seek to reorder memory accesses. Paek et al. [169] give a recent account of approaches for representing the results of array-access analysis. Kodukula et al. [134] use the Omega Calculator to enable transformations that better exploit memory hierarchies. It is unclear to me if optimizing numeric applications inherently requires more sophisticated arithmetic reasoning or if bounds-checks elimination has heretofore had less ambitious goals.

Most work on eliminating redundant checks on discriminated-union tags (equivalently, finding checks that might fail) has been for languages like Scheme [179] in which *all* values belong to one discriminated union. Eliminating checks is important for performance because every primitive operation (e.g., addition) must otherwise check type tags (e.g., that both operands are numbers). Wright and Cartwright [218] developed a practical "soft typing" implementation for Scheme. Soft typing is essentially type inference where there is enough subtyping that all programs remain typable. More precise types lead to fewer run-time tags and checks. Wright and Cartwright also summarize many other approaches, including

those based on flow analysis and abstract interpretation.

Another approach to approximating values that works well in languages like Scheme is set-based analysis. Flanagan's dissertation [71] investigates how to use such an analysis for a realistic language and how to avoid whole-program analysis techniques that inhibit scalability.

7.5.3 Languages

We now turn to languages that expose either the representation of arrays and unions or the checks associated with their safe use.

TALx86 [155, 96], an implementation of Typed Assembly Language [157] for Intel's IA-32 architecture, has support for using compile-time integers to describe data representation. Its array types, singleton-integer types, quantified types, and union types are essentially the assembly-language equivalent of the corresponding features in Cyclone. However, published work on TAL does not describe a system in which code producers can eliminate unnecessary bounds checks. Rather, macros are necessary for reading and writing array elements, and these macros always perform checks.

Unpublished work by David Walker eliminates this shortcoming to some extent. He tracks what this chapter calls compile-time constraints. Furthermore, a small proof logic lets programs prove results of the form $i < j$ subject to the assumed constraints. Compared to Cyclone, allowing proofs is more flexible than a flow analysis that basically encodes a restricted class of proofs. However, fixed-width arithmetic limits the collection of sound axioms. Walker's work also requires unpacking integers to singleton types before reasoning about their values. Though perhaps more pleasing from a type-theoretic standpoint than our flow analysis, it requires treating loops as polymorphic code. Cyclone's approach is probably more palatable for humans.

TALx86 also has union types. Annotations on conditional jumps guide the type system to refine the possible union members at the jump's destinations. The annotations are not essential, so it is unsurprising Cyclone does not need them.

Necula's proof-carrying code [161] provides a richer set of arithmetic axioms for programs to prove array-bound lengths. The compilers producing such code use theorem provers to eliminate checks as necessary. The instantiations of proof-carrying code I am aware of have all dictated data representation of arrays and discriminated unions as part of the policy. So while there is a richer language for eliminating checks, there is a weaker language for describing data.

Most relatedly, Xi et al. have used a restricted form of dependent type to reason about array lengths and unions in ML [224, 225, 221], a typed assembly language [223], and an imperative language called Xanadu [222]. In full generality,

dependent types are indexed by terms. But that means the undecidability of term equality (does e_1 equal e_2) make type equality undecidable. Therefore, Xi uses a separate language of type-index expressions and connects this language to terms via singleton types. This chapter does essentially the same thing; I have simply eschewed the terminology "dependent type" because I find it misleading when the syntax of types does not actually include terms.

Terminology aside, Xi's systems have compile-time integers, quantified types, and type constructors like Cyclone. The constraint language is more sophisticated, including quantification and many arithmetic operators. It is restricted to linear equalities that a variant of Fourier variable elimination can solve, but this restriction is only for compile-time efficiency. Xi has used integers to express invariants beyond array lengths and union members. Examples include the length of a linked list and the balance properties of red-black trees. For the former, the constraint language is expressive enough to express that an append function takes lists of lengths i and j and returns a list of $i + j$. Programmers must write some explicit loop invariants (or tolerate run-time checks), but Xi has developed significant type-inference techniques.

Xi's work on imperative languages [223, 222] shares some technical similarities with some work in this dissertation, but it is significantly less C-like. The formalism for Xanadu has *reference variables* that can change type, much like unescaped variables (as described precisely in Chapter 6) can change abstract rvalue. Indeed, both systems have typing judgments that produce typing contexts. However, Xi does not allow pointers to reference variables. In some sense, he treats escapedness as an invariant—all variables are either unaliasable or type invariant. So the technical contribution in Cyclone is support for statically tracking state changes for aliased objects so long as they are unescaped. As in C, we eliminate any run-time distinction between variables and heap addresses.

Another difference is that Cyclone supports mutating existential types whereas Xi's work has considered only pointers to existential types. Chapter 3 investigated the ramifications of this decision in great detail. Cyclone considers the avoidance of unnecessary levels of indirection a hallmark of C-style programming.

As a matter of emphasis, Xi has been less interested in user-defined data representation than in proving run-time checks cannot fail. For example, the formalism for Xanadu assumes a primitive operation for acquiring the length of an array given only the array. His work on safe assembly language supports the common implementation trick that pointers are usually distinguishable from small integers (as does the TALx86 implementation [155]). Supporting this trick in Cyclone should be possible, but it requires existential quantification over a single word that is either a small integer or a pointer.

Chapter 8

Related Languages and Systems

This dissertation describes a safe low-level programming language that has an advanced type system and a sound flow analysis. On some level, this endeavor relates to any work on program correctness, program semantics, program analysis, or language design. Furthermore, topics such as memory management, multithreading, and efficient array manipulation are well-studied areas with decades of research results. For such topics, the appropriate chapters present related work.

In contrast, this chapter takes a macroscopic view at prior and concurrent work on safe low-level programming. We focus on just the most closely related work and how Cyclone differs. Rather than enumerate research projects and their contributions, we categorize the projects. Like Cyclone, many projects employ a combination of approaches, so the categorization is only an approximation.

We begin with programming languages (other than C) that have relevant support for low-level systems. This discussion includes two "industrial-strength" languages (Ada and Modula-3) and some research prototypes. We then describe approaches for describing data representation (e.g., foreign-function interfaces). Section 8.3 contrasts Cyclone with lower-level safe languages, such as Typed Assembly Language. Section 8.4 describes systems that use unconventional data representation and memory management to implement C safely. Finally, Section 8.5 briefly surveys other compile-time approaches for checking safety properties, including theorem proving, model checking, type qualifiers, dependent types, pointer logics, and user-defined compiler extensions.

8.1 Programming Languages

This section contrasts some safe or almost-safe programming languages with support for controlling data representation or resource management.

Ada: Ada is a general-purpose programming language with substantial support for modularity, abstraction, concurrency, and user-defined data representation [194, 19]. Compared to Cyclone, it is less safe and at a higher level of abstraction. Ada is a big language with many relevant features; we do not discuss them all.

Ada has "escape mechanisms" for performing unsafe operations, such as memory deallocation. Ada also does not enforce that memory is initialized before it used; behavior is undefined when this error occurs. In Cyclone, the "escape mechanism" is to write part of the application in C.

The safe subset of Ada relies almost entirely on (optional) garbage collection for memory management. The exception is "limited types." In Cyclone terms, programmers can declare that objects for some type are allocated from a fixed-size region (the programmer picks the size). The region is deallocated when control leaves the scope of the type declaration. A run-time failure occurs if the program allocates too many objects of the type. Cyclone does not fix the size of regions (a simple extension could do so) nor does it conjoin the notion of type and lifetime.

Ada's "generics" allow polymorphic code, like ML functors, CLU clusters, or C++ templates. Generics are second-class constructs; their instantiation occurs at compile-time. As with C++ templates, conventional implementations generate code for each distinct instantiation. Chapter 3 explains that this technique produces more code but avoids unnecessary levels of indirection for program data.

Ada's "packages" are modules that support hiding code, data, and type definitions. One Ada feature would prove useful in Cyclone: types can have "private" fields. The size and alignment of these fields is exposed to other packages, but code in other packages still cannot use the fields. This technique allows other packages to allocate objects of the type and access other fields in the type efficiently. However, it prevents separate compilation. If the implementation of a type with private fields changes, it is necessary to recompile packages using the type.

Ada lets programmers specify the size (in bits) and order of record fields and numeric types. There is no support for the user specifying the location of data necessary for safety, such as array bounds or discriminated-union tags.

Modula-3: Modula-3 is a general-purpose programming language that rigidly distinguishes safe and unsafe modules [106]. The former cannot depend on the latter, thus placing less trust in unsafe modules than Cyclone would place in linking against C code. Code in unsafe modules may perform unsafe operations. Modula-3 uses an object-oriented paradigm for code reuse. The implementation controls the data representation for objects. However, Modula-3 also has records and numeric types of user-specified size.

Modula-3 was the implementation language for the SPIN extensible operating system [22], proving by example that Modula-3 is useful for writing untrusted systems extensions [185]. The SPIN implementors identified three language extensions they considered essential for their task [121]. First, they allowed casting arrays of bits to appropriate record types that contain no pointers. Cyclone has this ability. Second, they require some untrusted code to be ephemeral, meaning the system can safely terminate such code at any time. The compiler checks that ephemeral code does not perform inappropriate operations, such as allocate memory. Cyclone has no such notion; there is no language support for systems conventions like transactions. Third, they have first-class modules for dynamic linking and system reconfiguration. Cyclone has no intralanguage support for linking.

The SPIN project reports tolerable overhead from garbage collection, but they resort to coding conventions such as explicit buffers to reduce reliance on the collector. The language does not ensure these extra conventions are followed correctly.

Low-level services in SPIN, such as device drivers, are written in C. For language interoperability, the Modula-3 compiler produces C interfaces for Modula-3 types. Furthermore, data allocated by Modula-3 must be visible to the garbage collector even if the only remaining references to it are from C code.

Systems Programming in High-Level Languages: Although this dissertation presupposes that implementing operating systems and run-time systems benefits from controlling data representation and resource management, several research projects have nonetheless performed these tasks with high-level languages. These systems benefit from using safe languages, but they often require unsafe extensions and then try to minimize such extensions' use.

Operating systems implemented in Java [92] include J-Kernel [203] and KaffeOS [13]. The DrScheme programming environment [69] includes substantial support for running untrusted extensions much as operating systems manage untrusted user processes [78]. These systems address important requirements that Cyclone does not such as limiting resources (e.g., the amount of memory it can allocate) and revoking resources (e.g., aborting a process and recovering locks it holds). Back et al. compare the techniques for the Java systems [14]. Czajkowski and von Eicken describe JRes, the resource-accounting scheme underlying J-Kernel [53]. More recently, Hawblitzel and Von Eicken have taken a more language-based approach in the Luna system [111], in which the type system distinguishes revocable and irrevocable pointers.

The techniques developed in this dissertation appear ill-equipped to address this style of process-oriented resource control. Nonetheless, the OKE project [27] has modified Cyclone to ensure that untrusted kernel extensions are safe. In general,

using a safe language avoids the performance overhead of running all untrusted code in separate (virtual) memory spaces.

The Jikes Java Virtual Machine [125] is implemented entirely in Java, with a few extensions for reading and writing particular words of memory. Such extensions are necessary for services like garbage collection; they should not be available to untrusted components.

The Ensemble system uses OCaml to implement a flexible infrastructure for distributed-communication protocols [112, 113]. The developers provide substantial comparison with an early system written in C. They argue that safety and higher-level abstractions led to a smaller, more flexible, and more robust implementation with little performance impact. For one crucial data structure, garbage collection proved inappropriate so they resorted to explicit reference counting.

The Fox Project [109] uses Standard ML [149] for various systems tasks, such as network-protocol stacks. The project contends that safe languages and certified code (see Section 8.3) increase program reliability.

Vault: The Vault programming language [55, 66] uses a sound type system that restricts aliasing to ensure, at compile time, that programs use objects and interfaces correctly. By tracking the abstract state of objects, such as file descriptors, the type system can formalize interface protocols. For example, Vault can ensure that programs close files exactly once. The key technology is a type system in the spirit of the capability calculus [211] that ensures aliases to tracked objects are never lost. Extensions termed "adoption" and "focus" ameliorate the strong restrictions of capabilities without violating safety. Incorporating restricted aliasing into Cyclone is ongoing work.

Restricted aliasing allows safe use of explicit memory deallocation (like C's `free` function), allowing the Vault implementation to use no garbage collector. As such, it is easier to use Vault in environments such as operating systems that are hostile to automatic memory management. DeLine and Fähndrich have implemented a Windows device driver in Vault. The Vault interface to the kernel ensures the driver obeys several important protocols, such as not modifying interrupt packets after passing their "ownership" to other parts of the system.

In 2001, I implemented the same device driver in Cyclone. Compared to Vault, the Cyclone interface did not prevent several unsafe operations. In other words, the driver was memory safe, but it was still provided an interface through which it could crash the operating system. The lesson is that memory safety is necessary but not sufficient. On the other hand, Cyclone's C-level view of data representation was welcome. The Cyclone device driver had less than 100 lines of C code for performing operations inexpressible in Cyclone. In contrast, the Vault driver had over

2000 lines of C code, primarily for converting between Vault's data representation and C's data representation.

Vault and Cyclone are both research prototypes exploring powerful approaches to compile-time checking of low-level software. Although there is already some overlap (e.g., regions and type variables), much work remains to realize a smooth integration of the two approaches.

Cforall: The Cforall language [39, 57] attempts to improve C. In addition to syntactic improvements (e.g., pointer-type syntax) and additional language constructs (e.g., tuples and multiple return values), they have support for polymorphic functions. Compared to Cyclone, the project is more interested in remaining closer to C and less interested in ensuring safety.

Control-C: Control-C [135] combines severe restrictions on C with interprocedural flow analysis to ensure small (about 1000 lines) real-time control systems are safe without run-time checking or user annotations. The system disallows all pointer arithmetic and casts among pointer types, making it impossible to write generic code. Interprocedural analysis and an expressive arithmetic prove array-indexing is safe or reject a program. A primitive can deallocate *all* heap-allocated data (in this sense, there is one region) and the flow analysis ensures there are no dangling-pointer dereferences as a result. The designers claim this simple form of memory management suffices for small control applications. NULL-pointer dereferences and using uninitialized pointers are actually checked at run-time, but they use hardware protection and trap handlers to incur no performance overhead when the checks succeed. The work does not consider thread-shared data.

8.2 Language Interoperability

Implementations of high-level languages often provide access to code or data not written in the language. Such facilities require a way to describe a foreign function's argument types and foreign data's representation. Conversely, implementations often let C programs use code or data written in the high-level language. These interoperability mechanisms are related to Cyclone because a key requirement is an explicit definition of data representation at an appropriate level of abstraction. However, no projects I am aware of check the interoperability interface for safety.

Fisher, Pucella, and Reppy [70] explore the design space for foreign-function and foreign-data interfaces. Their compiler's intermediate language, BOL, is a low-level unsafe language rich enough to describe such interfaces. BOL is sufficiently powerful to allow their compiler infrastructure to implement cross-language

inlining. In fact, using BOL for interoperability is so lightweight that their infrastructure uses BOL to implement primitive operations, such as arithmetic.

Blume [24] provides Standard ML programs with direct access to data from C programs. His approach uses ML's type system to encode invariants about how the C data must be accessed. Compiler extensions provide direct access to memory.

Many systems use an IDL [100] (interface description language) to describe code and data without committing to a particular language. In fact, IDL uses rather C-like assumptions, but it has many interesting extensions. For example, an attribute can indicate that arguments of type `char*` are strings. Another example reminiscent of safety is an attribute specifying that one argument is the length of another (array) argument. Language implementations typically support IDL by generating "stub code" to mediate the mismatch between the implementation's internal data-representation decisions and an appropriate external interface.

Some languages specify an interface to C code without resorting to IDL. Perhaps the most well-known example is the Java Native Interface [142].

Allowing C code to access high-level language structures usually amounts to providing header files describing the implementation's data-representation decisions. More interesting are conventions for maintaining the run-time system's assumptions, such as the ability to find roots for garbage collection. One solution is to compile code from multiple languages to a common virtual machine [143, 28]. The virtual machine provides one run-time system for code from multiple source languages. Compilers can produce metadata to describe the data they produce. Virtual machines often assume security and resource-management obligations traditionally relegated to operating systems.

The C-- project [36, 131] is designing a language suitable as a target language for a variety of high-level language. C-- provides a more open run-time system than the virtual-machine approach. For example, the high-level language implementation can provide code that the run-time system uses to find garbage-collection roots. By extending the run-time system via call-back code in this way, C-- avoids a complicated language for describing data representation. In fact, types in C-- describe little more than the size of data, which is what one needs to compile a low-level language.

8.3 Safe Machine Code

Several recent projects have implemented frameworks for verifying prior to execution that machine code obeys certain safety properties. Verifying machine code lets us ensure safety without trusting a compiler that produces it or—in a mobile-code setting—the network that delivers it. This motivation leads to systems that

are substantially different than Cyclone in practice. First, because we expect most object code to be machine-generated (i.e., the result of compilation), safe machine languages are more convenient for machines than humans. In particular, expressiveness takes precedence over convenience and simplicity. Second, implementing a checker should be simpler than implementing a compiler. Otherwise, the framework does not reduce the size of the trusted computing base.

Typed Assembly Language (TAL) imposes a lambda-calculus-inspired type system on assembly code. Early work [157, 156] showed how compilers for safe source languages could produce TAL (i.e., machine code plus typing annotations). In particular, the type system can encode low-level control decisions such as calling conventions and simple implementations of exception handling. Later work explored how to use regions and linear types to avoid relying on conservative garbage collection for all heap-allocated data [211, 186, 212, 210]. An implementation for the IA-32 architecture included many important extensions [155], such as a kind system for discriminating types' sizes and a link-checker for safely linking separately compiled object files [89, 88]. I explored several techniques to reduce the size of type annotations and the time for type-checking TAL programs [96]. Compared to Cyclone, TAL has a much lower level view of control and a slightly lower level view of data. For the former, there is no notion in the language of procedure boundaries or lexical scope. For the latter, the language exposes the byte-level size and alignment of all data.

Proof Carrying Code (PCC) [161, 160] also uses annotations on object code to verify that the code meets a safety policy. By encoding the policies in a formal logic, policy designers can change the policy without changing the implementation of the checker. In practice, the policies that have been written cater to the calling conventions, procedure boundaries, and data representation of particular compilers, including a Java compiler [48] and a compiler for a small subset of C [162]. Compared to Cyclone or TAL, the policies have allowed more sophisticated proofs for eliminating array-bounds checks, but they cannot express memory-management invariants, aliasing invariants, or optimized data representations. Work on reducing the size of annotations and the time for checking [163, 165] focuses on eliding simple proofs and encoding proofs as directions for a search-based proof-checker to follow. These techniques make proofs smaller, but they are probably no more convenient (for machines or humans), so they make little sense for Cyclone.

Because TAL and PCC led to implementations with trusted components containing over 20,000 lines of code, other researchers have taken a minimalist or "foundational" approach to PCC [10, 9, 104]. In such systems, one trusts only the implementation of an expressive logic, an encoding of the machine's semantics in the logic, and an encoding of the safety policy. A compiler-writer could then prove (once) that a type system like TAL is sound with respect to the safety policy and

then prove (for each program) that the compiler output is well-typed. It is unclear if techniques from these projects could make the Cyclone implementation more trustworthy. Crary has encoded TAL-like languages in a formal metalogic [51]. Like foundational PCC, this project reduces the trusted computing base, but it is unclear how much the techniques apply to human-centric programming languages.

The minimalist approach has also tried to remove garbage collection from the trusted computing base. Powerful type systems supporting intensional type analysis [215] and regions can allow untrusted programmers to write simple garbage collectors [214, 153]. Cyclone is far too restrictive for writing a garbage collector, but the necessary typing technologies seem far too complicated for a general-purpose programming language.

Rather than using type-checking or proof-checking as the foundation for checking machine code, Xu et al. use techniques more akin to shape-analysis [181] and flow analysis [227, 228, 226]. Their approach requires no explicit annotations except at the entry point to untrusted code, but they use expensive program-verification techniques to synthesize induction invariants for loops and recursive functions. This approach allows checking code from unmodified compilers and compilers for unsafe languages, but it has been used only for programs with less than one thousand machine instructions. Furthermore, the type system (based on the physical type-checking work described in Section 8.5) and abstract model of the heap cannot handle existential types or discriminated unions. Rather, Xu's focus has been on inferring array sizes and the safety of array indexing. The interpretation of the program as modifying an abstract model of the heap captures more alias and points-to information than other approaches. Cyclone has stronger support for sophisticated data invariants, but less support for array-bounds and points-to information.

Kozen's Efficient Certifying Compilation (ECC) [136] tends to favor elegance, simplicity, and fast verification over the complex policies of the other frameworks. By exploiting structure in the code being verified, it can quickly and easily verify control-flow and memory-safety properties. Like the work in this dissertation, ECC checks code that separates operations like array-subscripts into more primitive steps like bounds-checks and dereferences. The focus in Cyclone has been on a type system expressive enough to allow programmers to move checks safely, such as hoisting them out of loops. It is usually straightforward to extend ECC to handle such optimizations.

8.4 Safe C Implementations

Contrary to many programmers' expectations, the C standard imposes weak requirements on an implementation's data representation and resource management. Therefore, an implementation can impose safety via run-time checks.

Pure Run-Time Approaches: Austin et al. developed such a system called Safe-C [12]. Instead of translating pointers to machine addresses, the implementation translates pointers to records including lifetime and array-bounds information. Each pointer dereference uses this auxiliary information to ensure safety. The program must also record the deallocation of heap objects and stack frames. Safe-C supports all of C, including tricky issues such as variable-argument functions. The disadvantages of this approach include performance (programs sometimes run an order of magnitude slower than with a conventional implementation) and data-representation changes (making it difficult to interface with unchanged code).

Jones and Kelly solve the latter problem by storing auxiliary information in separate tables [129]. That is, a pointer is again a machine address, but a pointer dereference first uses the address to look up the auxiliary information in a table.

McGary has developed an extension to the `gcc` compiler that allows pointers to carry bounds information and subscript expressions to check the bound [148].

Although not discussed in this dissertation, Cyclone has a form of pointer type that carries run-time bounds information. Such pointers permit unrestricted pointer arithmetic. Subscript operations incur a run-time check. Because Cyclone also has the compile-time approaches discussed in this dissertation, the programmer has the choice between the convenience of implicit run-time checks and the performance and data-representation advantages of unchanged code generation. However, Cyclone provides little support for run-time approaches to detecting dangling-pointer dereferences. The extension described in Section 4.4.2 is a partial solution, but it works for neither individual heap objects nor stack regions.

Several tools (Purify [110], Electric Fence [171], and StackGuard [50] are a few examples) also use run-time techniques to detect bounds violations and dangling-pointer dereferences. Techniques include using hardware virtual-memory protection to generate traps. For example, one way to detect dangling-pointer dereferences is to allocate each object on a different virtual-memory page and to make the page inaccessible when the object is deallocated. The performance costs of these tools mean they are primarily used for debugging. Different systems have different limitations. For example, Electric Fence detects only heap-object violations. One advantage is that tools can replace libraries or rewrite object code, which avoids recompilation.

Finally, software-fault isolation [209] provides sound, coarse-grained memory

isolation for untrusted components. It assigns such components a portion of the application's address space and then rewrites components' object code (typically with masking operations) to ensure all memory accesses fall within this space. This approach does not necessarily detect bugs nor is it appropriate for applications that share data across component boundaries, but it is simple and language-neutral.

Approaches Exploiting Static Analysis: If static analysis can prove that some run-time checks are unnecessary, then a system can recover many of Cyclone's advantages. In particular, there is less performance cost, less change of data representation, and fewer points of potential run-time failure. An automatic analysis is also more convenient. However, the programmer generally has less control than with Cyclone.

The CCured system [164, 38] uses a scalable whole-program static analysis in a safe implementation of C. It also can show programmers the analysis results to help with static debugging. The analysis is sound. Its essence is to distinguish pointer types so that not all of them have to carry bounds information. CCured has two kinds of pointer types that Cyclone does not: Sequence pointers allow adding nonnegative values to pointers (i.e., they permit unidirectional pointer arithmetic). Wild pointers let a pointer point to values of different types. The latter is important because CCured has no notion of polymorphism, although wild pointers are strictly more lenient than type variables. However, run-time type checks require run-time type information, which is not present in the Cyclone implementation. (Discriminated unions have tags, of course, but this information is not hidden from programmers.)

CCured relies on conservative garbage collection to prevent dangling-pointer dereferences. Programs where stack pointers are placed in data structures sometimes need to be manually rewritten. As described in Chapter 5, it is unclear how to extend CCured to support multithreading.

The main convenience of CCured over Cyclone is that programmers do not need to indicate which pointers refer to arrays (of potentially unknown length). Cyclone is less likely to accept unmodified C programs, but CCured does require some manual changes. For example, because CCured may translate different occurrences of a type `t` to different data representations, it forbids expressions of the form `sizeof(t)`. CCured also cannot support some valid C idioms, such as a local allocation buffer that creates a large array of characters and then casts pieces of the array to different types.

The CCured implementation ensures left-to-right evaluation order of expressions whereas Cyclone imposes no more ordering restrictions than C.

To summarize, in cases where the performance of a program compiled with

CCured is acceptable, the convenience over Cyclone makes CCured compelling for legacy code. Cyclone's language-based approach makes it easier for the programmer to control data representation and where run-time checks occur. The explicit type system may also make it easier to write new code in Cyclone.

Yong et al.'s Runtime Type-Checking (RTC) tool also can reduce performance cost by using static analysis [145, 229]. Some of the particular analyses they employ correspond closely to Cyclone features. These include ensuring data is initialized and ensuring pointers are not NULL. Unlike Cyclone, their analyses take as input a precomputed flow-insensitive points-to analysis, whereas Cyclone makes worst-case assumptions for "escaped" locations. This points-to information allows RTC to avoid redundant checks of repeated data reads when they can determine that no intervening mutation of the data is possible. Like CCured, RTC maintains run-time type information for all of memory (where it cannot be safely eliminated), but it does not store the information with the underlying data. RTC has no explicit support for threads.

8.5 Other Static Approaches

This section describes other projects that use compile-time techniques for ensuring software enjoys some safety properties. Most of the projects discussed analyze C programs. We address these questions for each project:

1. What properties are checked?
2. What assurances does the project give?
3. What techniques does the implementation use?
4. How does the project complement Cyclone's approach?

One point that we do not repeat for every (implemented) system is that these projects find real bugs in real software. (The projects in the previous section do too.) Empirical evidence is indisputable that many C programs, even those that have been used for years by many people, harbor lingering safety violations.

Physical Type Checking: Chandra et al. have developed a tool that checks type casts in C programs [184, 41]. They identify safe idioms for which C requires casts, including generic code (using `void*`) and simulating object-oriented idioms. For the latter, programs include "upcasts" and "downcasts" for casting to pointers to a prefix of the fields of a `struct` and vice-versa. They view τ_1* as a supertype of τ_2* when τ_1 describes a prefix of memory described by τ_2 (i.e., it is equivalent

to τ_2 except that it may be shorter). A novel constraint-based inference algorithm assigns types to expressions without consulting the actual types in the source program.

Empirical results show their tool can determine that about ninety percent of casts fit one of their sensible idioms. The remaining casts deserve close scrutiny. However, downcasts may not be safe. The tool does not consider other potential safety violations.

Cyclone supports the idioms identified to the extent safety allows. Chapter 3 discussed support for generic code. Subtyping (not discussed in this dissertation) uses a similar notion of physical layout. However, downcasts are not supported. Many object-oriented idioms can be avoided via other features (e.g., existential types and discriminated unions), but better object support remains future work. Because Cyclone is strongly typed, there is no reason to ignore the program's type annotations.

Type Qualifiers: Foster et al.'s work on "cqual" uses type qualifiers to enrich C's type system in a user-extensible way [80, 81, 82]. Whereas C has only a few qualifiers (`const`, `volatile`, `restrict`), cqual lets programmers define new ones. The qualifiers can enjoy a partial order (as examples, `const` is less than non-const and not-NULL is less than possibly NULL) and the system has qualifier polymorphism.

Interprocedural flow-sensitive analysis eliminates the need for most explicit annotations. In practice, programmers annotate only the key interface routines. For example, a qualifier distinguishing user pointers and kernel pointers helps detect security violations in operating-systems code. Functions that may produce user pointers and functions for which security demands they not consume user pointers require annotations. The system then infers the flow of user pointers in the program. Aliasing assumptions are sound and less conservative than in Cyclone.

The techniques in Cyclone cqual are complementary. Cyclone's focus on memory safety makes it less extensible than cqual. Without extensibility, we find ourselves extending the base language every time a new safety violation arises. On the other hand, cqual assumes the input program is valid ANSI C. That is, cqual is sound only if one assumes memory safety. The two systems do overlap somewhat. For example, both systems have been used to prevent NULL-pointer dereferences.

Extended Static Checking: ESC/Java [76] (and its predecessor ESC/Modula-3 [56]) uses verification-condition generation and automated theorem proving to establish properties of programs without running them. Although this checker

analyzes programs in a safe language, we compare it to Cyclone because it takes a quite different approach to eliminating similar errors. First, it identifies potential errors including NULL-pointer dereferences, array-bounds violations, data races, incorrect downcasts, and deadlocks. Second, it checks that the program meets partial specifications that users make in an annotation language.

ESC/Java translates Java to a simpler internal language, then generates a verification condition that (along with some axioms describing Java) must hold for the program to meet its partial specification, then uses a theorem prover to prove the verification condition, and finally generates warnings based on how the prover fails to prove the condition. This architecture involves more components than Cyclone, which is more like a conventional compiler with type-checking followed by flow analysis.

The ESC/Java implementation is neither sound nor complete. Incompleteness stems from the theorem prover (which is sound but operates over a semidecidable logic) and from modularity in the verification condition (which means abstraction can lead to a verification condition that is unnecessarily strong). Unsoundness stems from ignoring arithmetic overflow (to avoid spurious warnings) and from analyzing loops as though their bodies execute only once. The user can specify that ESC/Java should unroll loops a larger (fixed) number of times. The system treats loops with explicit invariants soundly.

Whereas Cyclone uses distinct syntax for terms (e.g., `x`) and compile-time values (e.g., `tag_t<`i>`), the annotation language for ESC/Java contains a subset of Java expressions. For programmers, reusing expressions is easier. However, it is less convenient from the perspective of designing a type system. (Dependent type systems use terms in specifications, but mutation complicates matters.)

To reduce programmer burden and spurious warnings, the Houdini tool [75] attempts to infer annotations by using ESC/Java as a subroutine. A similar approach for inferring Cyclone prototypes, using the Cyclone compiler as a subroutine, might prove useful.

Lint-Like Tools: The original Lint program [127] used simple syntactic checking to find likely errors not reported by early C compilers. More recently, more sophisticated tools implement the basic idea of finding anomalies in C code at compile-time.

LCLint [138, 63] and its successor Splint [189, 65] use intraprocedural analysis and (optional) user-provided function annotations to find possible errors and avoid false positives. A vast number of annotations give users control over the tool; any warning is suppressible for any block of code. Early work focused on ensuring code respected abstract-datatype interfaces and modification to externally visible

state (e.g., global variables) was documented [64]. Subsequent work focused on safety violations including NULL-pointer and dangling-pointer dereferences, as well as memory leaks. Pointer annotations include notions of uniqueness (there are no other references) and aliasing (a return value may be a function parameter). The expressive power of these annotations and Cyclone's region system appear incomparable, but they capture similar idioms. LCLint also warns about uses of uninitialized memory and has an annotation similar to Cyclone's "initializes" attribute (see Section 6.3).

LCLint is neither sound nor complete. In particular, its analysis acts as though loop bodies execute no more than once. It checks (unsoundly) for expressions that are incorrect because of C's under-specified evaluation order. Other parts of the tool then assume all orders are equivalent. Cyclone might benefit from this separation of concerns.

The Splint tool's primary extensions are support for finding potential array-bounds violations and support for allowing the user to define new checks. For arrays, function annotations describe the minimum and maximum indexes of an array that a function may access. The expression language for indexes includes arithmetic combinations of identifiers and constants. The tool uses arithmetic constraints and algebraic simplification to analyze function bodies. It does not appear that type definitions can describe where programs store array lengths. To analyze loops, Splint uses a set of heuristics to find common loop idioms. These idioms include pointer-arithmetic patterns that Cyclone does not support. Splint unsoundly assumes any bounds violation will occur in the first or last loop iteration because this simplification works well in practice.

Splint's extensibility allows programmers to declare new attributes and specify how assignments and control-flow joins should combine the attributes. The language is rich enough to track values, such as whether a string could be "tainted" via external input. The extension language appears much weaker than the *metal* language described below.

The PREfix tool [34] also finds program errors such as NULL-pointer dereferences, memory leaks, and dangling-pointer dereferences. It has been used with many commercial applications, comprising a total code base of over 100 million lines of code. PREfix expects no explicit annotations, so it is trivial to use. The primary challenge in implementing PREfix is avoiding spurious warnings because it must discover all static information not provided by C. PREfix attempts to find only a fixed collection of errors (not including, it appears, array-bounds errors). It is unsound and considers only one evaluation order for expressions.

PREfix ensures scalability by generating a model for each function and using the model at call sites. (It unsoundly evaluates recursive call cycles a small number of times, typically twice.) These models are quite rich: They can require properties of

parameters, produce results that depends on parameter values, and describe effects on memory (including global variables). Intraprocedurally, PREfix examines all feasible execution paths, up to a fixed limit to avoid taking time exponential in a function's size. Heuristics guide which paths to examine when there are too many. A rich language of relations and constraints among variables (e.g., `x > 2*y`) discovers infeasible paths, which is crucial for avoiding spurious warnings. A generic notion of "resource" tracks similar problems such as using freed memory or reading from a closed file handle.

PREfix produces excellent error messages, describing control paths and reasons for the warning. It also filters unimportant warnings, such as memory leaks in code that executes only shortly before an application terminates.

Cyclone and PREfix use very different techniques. PREfix is certainly more useful for large commercial applications for which nobody will modify the code or insert explicit annotations. Many of the errors it detects are impossible in Cyclone, and providing the annotations is straightforward when writing new code. Its detection of misused resources (leaks and use-after-revocation) is finer grained than Cyclone's support for resource management.

Metacompilation: Engler et al. have developed the *metal* language and *xgcc* tool, which allow users to write static analyses [102, 43]. The language has primitive notions of state machines and patterns for matching against language constructs. These features make it extremely easy to write analyses that check for idioms such as, "functions must release locks they acquire," or, "no floating-point operations allowed." The analysis is automatically implemented as a compiler extension (hence the term *metacompilation*). Simple application-specific analyses have found thousands of bugs in real systems [61]. The metal language allows analyses to execute arbitrary C code, so it is quite expressive.

For scalability and usefulness, Engler exploits many of the same unsoundnesses of the Lint-like tools. Examples include optimistics assumptions about memory safety, aliasing, and recursive functions. The checking is quite syntactic. For example, an analysis that forbids call to `f` could allow code that assigns `f` to a function-pointer variable `g` and then calls through `g`. Because we do not expect nonmalicious code to do such things, a bug-finding tool may not suffer from such false negatives.

The *xgcc* tool employs context-sensitive interprocedural and path-sensitive intraprocedural dataflow analysis. Although such analyses could take time exponential in the size of programs, such cost does not occur in practice: Aggressive caching of analysis results and the tendency for programs to have simple control structures (with respect to the constructs relevant to the analysis) are crucial.

When the tool finds thousands of potential bugs, it uses statistical techniques to rank which ones are most likely to be actual errors. If many potential violations arise along control-flow paths following a function `f`, it is likely they are false positives resulting from an imprecise analysis of `f`. Engler also uses statistics to infer analyses automatically [62]. Essentially, a tool can guess policies (e.g., calls to `g` must follow calls to `f`) and report potential violation only if the policy is followed almost all of the time. (This work is similar in spirit to work on mining specifications [5], but the latter uses machine-learning techniques to analyze run-time call sequences.) It could prove useful to use similar statistical techniques to control the sometimes impenetrable error messages from the Cyclone compiler, especially when porting C code. For example, if many errors follow calls from `f`, the compiler could suppress the errors and try to find a stronger type for `f`.

The extensibility that metacompilation provides is difficult to emulate within a language like Cyclone. Although clever programmers can sometimes write interfaces that leverage a type-checker to enforce other properties [79], application-specific idioms such as calling sequences remain difficult to capture. For example, the Cyclone compiler itself has an invariant that all types in an abstract-syntax tree have been passed to a `check_valid_type` function before the abstract-syntax tree is type-checked. Lack of automated support for checking this invariant has produced plenty of bugs. In short, metacompilation is a good complement to sound static checking.

Model Checking: A model checker ensures an implementation meets a specification by systematically exploring the states that the implementation might reach [47]. (Specifications are often equivalent to temporal-logic formulas, so exhaustive state exploration can establish that the implementation is a model of the formula, in the formal-logic sense.) Given the initial conditions, a model of the environment, and the possible state transitions, a model checker can search the state space. Upon encountering an error, it can present the state-transition path taken. Compared to conventional testing, model checking achieves greater coverage by not checking the same state twice. Compared to flow analysis and type systems, model checking is more naturally path sensitive.

Model checking, even model checking of software, is too large a field to describe here, so we focus only on projects that model check C code. (To contrast, many systems require a human to abstract the software to a state machine. Checking this abstraction can catch some logical errors, but not necessarily implementation errors.) The challenge of software model checking is the "state-explosion problem." Typical systems have too many distinct states (perhaps infinite) for an efficient checker to remember which states have been visited.

VeriSoft [90] and CMC [159] model check C implementations of event-handling (i.e., reactive) code by actually running the code with a scheduler that induces systematic state exploration. Programmers must provide the specification to check, the entry points (event handlers) for the code, and a C implementation of the assumed environment. VeriSoft runs different system processes in different host-operating-system processes. Therefore, safety violations do not compromise the model. CMC uses only one operating-system process. Both systems *assume* that the C implementations of a process make deterministic, observable transitions. That is, the model checker assumes the C code terminates and does not have internal nondeterminism. A safety violation presumably leads to nondeterminism (e.g., reading arbitrary memory locations to compute a result). However, CMC does detect some safety violations such as dangling-pointer dereferences and memory leaks.

SLAM [17, 16] and BLAST [117, 116] automatically create a model that soundly abstracts a C program, and then model check the abstract model against the user-provided specification. If the abstract model does not meet the specification, the model checker creates a counterexample. A theorem prover then determines if the counterexample applies to the C program or if it results from the model being too abstract. For the latter, the system automatically generates additional predicates that induce a less abstract model to model check. This architecture is known as an abstract-verify-refine loop. It is sound (each abstract model is a correct abstraction of the code) and complete (it does not report counterexamples unless they apply to the code), but the process may not terminate. Furthermore, both systems assume the C program has no array-bounds violations and no order-of-evaluation errors. The theorem provers assume there is no arithmetic overflow.

BLAST uses "lazy abstraction" for efficiency. It does not completely rebuild the abstract model on each refinement iteration. Instead, the additional predicates induce a less abstract model only for the parts of the model relevant to the counterexample. BLAST has been run on programs up to sixty thousand lines. Both systems have been used to verify (and find bugs in) device drivers of several thousand lines.

Finally, Holzmann's AX [119] and Uno [120] tools use model-checking techniques to check C programs. The former assumes programs are ANSI-C. It extracts a model and represents it with a table. Users can then modify the table to interpret certain operations correctly (e.g., function calls that are message sends). This framework does not preclude using tools to ensure the modification is sound, but the focus is extracting most of the model automatically. Uno is more like *metal* or *lint* (see above). By default, it looks for uses of uninitialized variables, array-bounds violations (for arrays of known size), and NULL-pointer dereferences. It uses model-checking techniques for intraprocedural analysis. Therefore, it can

exploit more path-sensitivity than Cyclone. It does not appear that there is much support for nested pointers and data structures. Uno lets programmers write new checks using C (which Uno then interprets) enriched with ways to match against definitions and uses in the program being checked.

Compared to Cyclone, model checking is superior technology for checking application-specific temporal properties. The projects described here demonstrate that it is feasible for medium-size C programs of considerable complexity. The generated counterexamples are useful for fixing bugs. Although the systems are sound with respect to some safety violations (e.g., incorrect type casts), there remain caveats (e.g., array bounds). Furthermore, model checking remains slower in practice than type-checking. Therefore, like metacompilation, the technologies seem complementary: A sound type system for detecting safety violations makes safety an integral result of compilation, but model checkers can check more interesting properties and are less prone to false positives. It would be interesting to enrich Cyclone with more path-sensitive checking, using model-checking techniques to control the potentially exponential number of paths.

Dependent Types: Section 7.5 describes Xi et al.'s work on using dependent types for soundly checking properties of low-level imperative programs [221, 223, 222]. They argue that for imperative languages with decidable type systems, it is important to make a clear separation between term expressions and type expressions. As such, the difference between "dependent types" and Cyclone's approach is little in principle. However, Xi's systems use more expressive compile-time arithmetics. Integrating such arithmetics and explicit loop invariants into Cyclone should pose few technical problems, but it may not be simple.

Cleanness Checking: Section 6.5 describes work by Dor et al.[58, 59] to use shape analysis and pointer analysis to check for some C safety violations. They call the absence of such violations "cleanness." The sophisticated analyses they use are sound and produce more precise alias information than Cyclone. They also check that no memory becomes unreachable. Current work to enrich Cyclone with unaliased pointers may achieve some of these goals, but with less precision.

As discussed in Section 7.5, Dor has also used an integer analysis to detect misuse of C strings [60]. Work to extend the approach to all of C is ongoing. This work appears to confirm the experience in Cyclone that the important abstractions for ensuring safe arrays describe the values of index expressions. Section 7.5 also describes work by Wagner et al. [208, 207] that uses an abstraction of C string-library functions and unsound aliasing assumptions to find bugs.

Property Simulation: Das et al.'s ESP project [54] uses a technique they call property simulation to make path-sensitive verification of program properties scalable. They seek to verify properties similar to those checked with model-checking tools like SLAM while enjoying the scalability of interprocedural dataflow analysis.

One way to distinguish model checking and conventional flow analysis is to consider their treatment of control-flow merge points. Whereas model checking maintains information from both paths to the merge, flow analysis soundly merges them into one abstract state. The key insight in ESP is to use the property being checked to strike a middle ground: Viewing the property as a finite-state machine (with an "error" state to indicate the property is not met), ESP merges abstract states for and only for paths for which the finite-state machine is in the same state.

Property simulation takes as input a global alias analysis as input, so most soundness issues are relegated to this preceding phase.

ESP gains efficiency by checking one property at a time and using the definition of the property to guide the precision of its analysis. In contrast, the Cyclone compiler checks for all safety violations at the same time. If Cyclone were to incorporate more path-sensitivity, it might become faster to check properties independently.

Chapter 9

Conclusions

In this dissertation, we have designed and justified compile-time restrictions (in the form of a type system and a flow analysis) to ensure that Cyclone is a safe language. Techniques such as quantified types and must points-to information have allowed the resulting system to achieve significant expressiveness. We have used similar approaches to solve several problems, including incorrect type casts, dangling-pointer dereferences, data races, uninitialized data, NULL-pointer dereferences, array-bounds violations, and misused unions. This similarity supports our thesis that the system is safe and convenient.

In this chapter, we summarize the similarities of our solutions and argue that they capture a natural level of expressive power. We then describe some general limitations of the approaches taken. Section 9.3 describes the implementation status of this dissertation's work and experience using the Cyclone implementation. Finally, Section 9.4 briefly places this work in the larger picture of building safe and robust software systems.

9.1 Summary of Techniques

Type Variables: We use type variables, quantified types, type constructors, and singleton types to describe data-structure invariants necessary for safety.

These invariants describe one data value that is necessary for the safe manipulation of another data value. The former can be an integer describing the latter's length, a lock that ensures mutual exclusion for the latter, a region handle describing the latter's lifetime, or a value of an unknown type that must be passed to code reachable from the latter. In all cases, we use the same type variable for both data values—what changes is the kind of this type variable.

To bind type variables, we use universal quantification, existential quantifica-

tion, or a type constructor. Universal quantification allows code reuse. Existential quantification lets us define data structures that do not overly restrict invariants. For example, one field can be the length of an array that another field points to, without committing to a particular length. Type constructors let us reuse data descriptions for large data structures. For example, we could define a list of arrays, where all the arrays have the same length. Together, existential quantification and type constructors let programmers enforce invariants at a range of granularities.

Singleton types (types for which only one value has that type) prove useful for regions, locks, and compile-time integers. In Cyclone, no two region handles have the same type, no two locks have the same type, and no distinct integer constants have the same `tag_t` type. Singletons ensure the typing rules for primitive constructs are sound. For example, if two region handles have the same type, then the type we give to an allocation expression (`rnew(r,`e`)`) could imply a lifetime that is too long. Similarly, if a test concludes that a value of type `tag_t<`i`>` is greater than 37, then we do not conclude the inequality for the "wrong" constant.

Effects and Constraints: Whereas data structures often enjoy safety-critical invariants, whether code can safely access data often depends on the program point. Effects summarize what is safe at a program point and what is necessary to call a function. (For the former, we sometimes call the effect a capability.) Examples include the names of held locks, the names of live regions, and inequalities among compile-time constants. Run-time operations influence effects. For example, acquiring a lock before executing a statement increases the effect for the statement. Similarly, tests between integers can introduce inequalities along for the succeeding control-flow branches.

Using effects as function preconditions keeps type-checking intraprocedural. The type-checker ensures call sites satisfy the effect and assumes the effect to check the function. However, if our effect language includes only "regions live," "locks held," etc., then the type system is too restrictive for polymorphic code. We cannot say that a function that quantifies over a type variable α is safe to call if the call instantiates α with a type that describes only live data. Therefore, we have effects that describe this situation and the analogous one for locks.

For existential types, we need some way to describe an abstract type's lifetime or locks without reference to a particular call site. To solve this problem, we have constraints that bound an effect with another one, e.g., locks(α)$\subseteq \{\ell\}$. These constraints also prove useful for describing outlives relationships for regions and preconditions for functions using a callee-locks idiom.

Prior work integrating effects and type variables used abstract effects instead of constraints and effects of the form locks(α). We have shown that abstract

effects are less convenient for user-defined abstract data types and that we can use Cyclone's effects to simulate abstract effects.

Flow Analysis: For safety issues for which data-structure invariants prove insufficient, we use a sound flow analysis that is more flexible than a conventional type-system approach. Problems include dereferences of possibly-NULL pointers, uninitialized data (the only safe invariant would outlaw uninitialized data), and mutable integers (which are necessary for loops that use the same variable to access different elements of an array).

For each program point, the analysis assigns each "abstract location" (roughly corresponding to a local variable or allocation point) an "abstract value." The abstract value soundly approximates whether any actual value it represents may be uninitialized, NULL, or within some integer intervals. However, because pointers and aliasing are so pervasive in C programs, soundness and expressiveness require the flow analysis to maintain some pointer information, as we summarize below.

Our analysis is intraprocedural, but additional function preconditions can capture some common interprocedural idioms. Intraprocedurally, the analysis takes the conventional flow-analysis approach of checking a program point under a single abstract state that soundly approximates the abstract states of all control-flow predecessors. That is, the analysis is path-insensitive.

Must Points-To and Escapedness Information: Because C's address-of operator lets programs create arbitrary aliases to any location, a sound flow analysis cannot assume that aliases do not exist. However, it is sound to assume aliases do not exist when memory is allocated (either by a variable declaration or by a call to `malloc`). Furthermore, tracking pointer information is necessary for separating the allocation and initialization of data in the heap.

To handle these issues in a principled way, Cyclone's flow analysis includes must points-to information and makes worst-case assumptions for locations for which not all aliases are known. That is, at a program point, we may know that abstract location x must hold a pointer to abstract location y. In particular, the analysis result for a `malloc` expression is that the result must point to the abstract location representing the allocation.

To enforce soundness in the presence of unknown aliases, the analysis maintains whether there may be a pointer to an abstract location that is not tracked with must points-to information. If so, the pointed-to location must have an abstract value determined only from its type. In particular, it must remain initialized and possibly NULL (unless its type disallows NULL pointers). It is a compile-time error to create an unknown alias of uninitialized memory.

9.2 Limitations

Because it is undecidable if a C program commits a safety violation, sound compile-time techniques necessarily reject safe programs. This section describes some general sources of approximation and possible approaches to relax them.

Data-Structure Invariants: Although the combination of existential types and type constructors gives programmers considerable power for describing invariants, restrictions on the scope of type variables is a limitation. For example, consider a thread-shared linked list of integers: With existential types and a lock field at each list node, we can describe a list where each integer is guarded by a (possibly) different lock. With a type constructor, we can describe a linked list where every integer is guarded by the same lock. Using both techniques, other invariants are possible. We can describe lists where the "odd" list positions (first, third, ...) use one lock and the "even" list positions (second, fourth, ...) use another. We can also describe lists where the first three elements use one lock, the second three another lock, and so on.

However, it is impossible to describe an invariant in which we have two lists, one for locks and one for integers, in which the i^{th} lock in one list guards the i^{th} integer in the other list. Similarly, if the integers are in an array, no mechanism exists for using two locks, each for half of the array elements. It does not appear that type-theoretic constructs extend well to support such data structures, which unfortunately threatens our thesis that Cyclone gives programmers control over data representation. One possible way to overcome this gap is to prove that an abstract data type is equivalent to one for which we can use standard typing techniques, but developing an appropriate proof language may prove difficult.

Programmers also cannot express certain object-oriented idioms within Cyclone's type system. Although recursive and existential types permit some object encodings, they prove insufficient for some optimized data representations and advanced notions of subtyping [33, 1, 88]. It may suffice to extend Cyclone with additional forms of quantified types.

Type Inference and Implicit Downcasts: To reduce the programmer burden that Cyclone's advanced type system imposes, we use intraprocedural type inference and some implicit run-time checks. Given an explicitly typed program, where the checks occur is well-defined, so programmers maintain control. For example, when dereferencing a possibly-NULL pointer or assigning a possibly-NULL pointer to a not-NULL pointer, the compiler can insert a check (and warn the programmer). Unfortunately, it is not clear how to infer types and implicit checks in a

principled way, as this example demonstrates. (The keyword `let` indicates that the type-checker should infer a declaration's type.)

```
void f(int b, int* q) {
  let p = new 0;
  p = q;
  printf("x");
  *p = 37;
}
```

If `q` is NULL, then either `p=q` or `*p=37` should fail its run-time check. However, which assignment should fail depends on the type of `p`. If `p` has type `int*`, then `*p=37` fails. If `p` has type `int@`, then `p=q` fails. It is undesirable for the result of type inference to affect program behavior, but it is unclear what metrics should guide the inference. In our example, either choice leads to one inserted run-time check.

Aliasing: Although Cyclone's flow analysis tracks must points-to and escapedness information, programmers cannot provide annotations to describe stronger points-to information. In particular, the flow analysis can track information only up to a pointer depth (the depth being the number of pointer dereferences needed to reach the location) that depends on the number of allocation sites in a function. For escaped locations, which includes all locations with too large a pointer depth, Cyclone assumes the most pessimistic aliasing assumptions.

A language for expressing stronger alias information would make Cyclone more powerful. For example, C99 [107] has the `restrict` type qualifier to indicate that a function parameter points to a location that the function body cannot reach except through the parameter. Work is underway to add "unique" pointers to Cyclone. A unique pointer must be the only pointer to the pointed-to location [32]. A type distinction between unique and nonunique pointers distinguishes the uniqueness invariant. If an unescaped location holds a unique pointer, it is safe to treat the pointed-to location as unescaped. But unlike Cyclone's flow analysis, adding unique pointers to the type system lets programmers express uniqueness invariants for unbounded data structures, such as linked lists. Unique pointers also permit manual memory deallocation and the safe "migration" of exclusive access to another thread.

Unique pointers are not more general than `restrict` because the latter can permit unknown aliases provided that those aliases are unavailable in some scope. This distinction illustrates that a static system can define a virtual partition of the heap and require that all unknown pointers to a location reside in one part of the

partition. This idea underlies the focus operator in the Vault language [66] and has been investigated foundationally in the logic of bunched implications [122].

Finally, we reiterate the discussion in Chapter 6 distinguishing alias information and must points-to information. Without the former, we cannot accept code that is safe only because `x` and `y` point to the same location unless we actually know which location.

Relations: The type and flow information in Cyclone is all "point-wise" meaning the information for each location is independent. For example, we may know `x` is not NULL, but there is no way to express that `y` is not NULL only if `x` is not NULL. The lack of aliasing information is another example.

Arithmetic and Loops: As discussed in Chapter 7, Cyclone supports little compile-time arithmetic. Therefore, if `len` holds the length of an array `arr`, we can accept `if(x<len) arr[x];` (assuming `x` is unsigned), but not `if(x<len && x > 1) arr[x/2 - 1];`. Many loops are safe only because of nontrivial properties of fixed-width arithmetic.

Cyclone also has no support for loop invariants that describe parts of arrays. This limitation often makes it impossible to check that loops correctly perform operations such as initialize all elements of an array.

Resource Exhaustion: We have not discussed safety violations that result from consuming excessive resources. For example, programs that open too many files or allocate too much memory can exceed limits set by the operating system or the underlying hardware. In the Cyclone implementation, we handle such problems by having the routines that open files or allocate memory throw exceptions when the resource is exhausted. There is no static support for ensuring programs do not exceed resource bounds. This omission makes Cyclone less useful for embedded systems.

Worse, the current Cyclone implementation does not detect stack overflow. That is, too many nested function calls or stack-allocated objects can exhaust the memory for the control-flow stack. This situation can lead to an illegal memory access (segmentation fault) or even a corruption of heap-allocated data. Unfortunately, it is difficult to detect stack overflow without hurting the performance of function calls.

9.3 Implementation Experience

This dissertation does not present empirical results from the Cyclone implementation. Such results do exist. Quantitative measurements regarding the code changes necessary to port C code to Cyclone and the run-time performance implications suggest that Cyclone is a practical choice [126, 97]. Furthermore, subsequent changes to Cyclone have led to substantial improvements. In particular, some benchmarks spend most of their time in loops that iterate over arrays. Support for allowing users to hoist array-bounds checks out of loops (even when the array length is not a compile-time constant) significantly improves the results for these benchmarks. Techniques described in Chapter 7 made this improvement possible.

In this section, we make some brief qualitative observations based on Cyclone programming experience that are relevant to the work in this dissertation. First, Cyclone's support for quantified types provides excellent support for generic libraries such as linked lists and hashtables. In practice, using such libraries is simple. Callers do not have to instantiate type variables explicitly nor do they need to cast results.

Second, Cyclone's region-based memory management is quite practical. The compile-time guarantee that stack pointers are not misused actually makes it more convenient to use stack pointers than in C. Growable regions make it simple to use idioms where callers determine object lifetimes but callees determine object sizes. However, simply using a garbage-collected heap is often as fast or faster than using growable regions.

Third, many applications, such as most of the Cyclone compiler, do not need control over data representation. Although not emphasized in this dissertation, Cyclone provides built-in support for arrays that carry run-time size information and discriminated unions that carry run-time tags. Use of these built-in features pervades Cyclone code. In particular, Cyclone supports full pointer arithmetic only for pointers (arrays) that have implicit bounds fields. Nonetheless, the work in this dissertation helped design these built-in features. For example, the problem with mutable existential types discussed in Chapter 3 applies to mutable discriminated unions, even if the programmer does not choose the data representation.

Fourth, the support for multithreading is not yet implemented. Although its similarity with other features suggests that we can implement a practical system with a compile-time guarantee of freedom from data races, such a system does not exist yet.

Fifth, support for separating the allocation and initialization of data is mostly successful. Occasionally the lack of path-sensitivity in the flow analysis forces programmers to include unnecessary initializers. Much more problematic is the lack of support for initializing different array elements separately. For arrays that

hold types for which 0 or NULL are legal values, using `calloc` in place of `malloc` proves useful (which Cyclone supports), as would implicit initialization of stack-allocated arrays (which Cyclone does not do). In practice, this problem has led the implementation not to check for initialization of numeric data. This compromise ensures the compiler never rejects programs due to uninitialized character buffers. Although we may miss bugs as a result, correct initialization of characters is unnecessary for memory safety.

9.4 Context

This dissertation develops an approach for ensuring that low-level systems written in a C-like language enjoy memory safety. The Cyclone language is a proof-of-concept for using *a rich language of static invariants and source-level flow analysis to provide programmers a convenient safe language at the C level of abstraction.* As this chapter has summarized, Cyclone's compile-time analysis is an approximate solution to an undecidable problem. It uses a small set of techniques to give programmers substantial low-level control, but significant limitations remain.

However, memory safety for a C-like language is just one way to help programmers produce better software. Memory safety does not ensure correctness. At best, it can help isolate parts of a software system such that programmers can soundly use compositional reasoning when building systems. Safety and compositional reasoning are just a means to an end. We are more interested in correct software, or at least in software that has certain properties such as security (e.g., not leaking privileged information).

Hopefully a memory-safe low-level language can provide a suitable foundation on which we can build assurances of higher-level properties. After all, enforcing such properties without memory safety is impossible in practice.

There is little hope that we will rewrite the world's software in Cyclone, and there are good reasons that we should not. Different programming languages are good for different tasks. Large systems often comprise code written in many languages. Even if each language is safe under the assumption that all code is written in that language, incorrect use of "foreign" code can induce disaster. Although Cyclone is a useful "brick" for creating robust systems, I hope future programming-languages research focuses more on the "mortar" that connects code written in different languages, compiled by different compilers, and targeted for different platforms.

Appendix A

Chapter 3 Safety Proof

This appendix proves Theorem 3.2, which we repeat here:

Definition 3.1. *State $H; s$ is stuck if s is not of the form* return v *and there are no H' and s' such that $H; s \xrightarrow{s} H'; s'$.*

Theorem 3.2 (Type Safety). *If $\cdot; \cdot; \cdot; \tau \vdash_{\text{styp}} s$, $\vdash_{\text{ret}} s$, and $\cdot; s \xrightarrow{s}{}^{*} H'; s'$ (where $\xrightarrow{s}{}^{*}$ is the reflexive, transitive closure of $\xrightarrow{s}$), then $H'; s'$ is not stuck.*

Like all proofs in this dissertation, the proof follows the syntactic approach that Wright and Felleisen advocate [219]. The key lemmas are:

- Preservation (also known as subject reduction): If $\vdash_{\text{prog}} H; s$ and $H; s \xrightarrow{s} H'; s'$, then $\vdash_{\text{prog}} H'; s'$.
- Progress: If $\vdash_{\text{prog}} H; s$, then s has the form return v or there exists H' and s' such that $H; s \xrightarrow{s} H'; s'$.

Given these lemmas (which we strengthen in order to prove them inductively), the proof of Theorem 3.2 is straightforward: By induction on the number of steps n taken to reach $H'; s'$, we know $\vdash_{\text{prog}} H'; s'$. (For $n = 0$, we can prove $\vdash_{\text{prog}} \cdot s$ given the theorem's assumptions by letting $\Gamma = \cdot$ and $\Upsilon = \cdot$. For $n > 0$, the induction hypothesis and the Preservation Lemma suffice.) Hence the Progress Lemma ensures $H'; s'$ is not stuck.

Proving these lemmas requires several auxiliary lemmas. We state the lemmas and prove them in "bottom-up" order (presenting and proving lemmas before using them and fixing a few minor omissions from the proofs in previous work [93]), but first give a "top-down" description of the proof's structure.

Preservation follows from the Term Preservation Lemma (terms can type-check after taking a step) and the Return Preservation Lemma (evaluation preserves

$\vdash_{\text{ret}} s$, which we need to prove the Term Preservation Lemma when the term is $\mathsf{call}\ s$). Progress follows from the Term Progress Lemma. The Substitution Lemmas provide the usual results that appropriate type substitutions preserve the necessary properties of terms (and types contained in them), which we need for the cases of the Term Preservation Lemma that employ substitution. The Canonical Forms Lemma provides the usual arguments for the Term Progress Lemma when we must determine the form of a value given its type.

Because the judgments for terms rely on judgments for heap objects (namely get, set, and gettype), the proofs of Term Preservation and Term Progress require corresponding lemmas for heap objects. The Heap-Object Safety Lemmas are the lemmas that fill this need. Lemmas 1 and 2 are quite obvious facts. Lemma 3 amounts to preservation and progress for the get relation (informally, if gettype indicates a value of some type is at some path, then get produces a value of the type), as well as progress for the set relation (informally, given a legal path, we can change what value is at the end of it). We prove these results together because the proofs require the same reasoning about paths. Lemma 5 amounts to preservation for the set relation. The interesting part is showing that the $\vdash_{\text{asgn}}$ judgment preserves the correctness of the Υ in the context, which means no witnesses for &-style packages changed. Given $\text{set}(v_1, p, v_2, v_1')$, Lemma 5 proves by induction the rather obvious fact that the parts of v_1' that were in v_1 (i.e., the parts not at some path beginning with p) are compatible with Υ. Lemma 4 provides the result for the part of v_1' that is v_2 (i.e., the parts at some path beginning with p). Reference patterns significantly complicate these lemmas; see the corresponding lemmas in Chapter 3 to see how much simpler the lemmas are without reference patterns.

The Path-Extension Lemmas let us add path elements on the right of paths. We must do so, for example, to prove case DL3.1 of Term Preservation. The proofs require induction because the heap-object judgments destruct paths from the left.

The remaining lemmas provide more technical results that the aforementioned lemmas need. The Typing Well-Formedness Lemmas let us conclude types and contexts are well-formed given typing derivations, which need to do to satisfy the assumptions of other lemmas. It is uninteresting because we can always add more hypotheses to the static semantics until the lemmas hold. The Commuting Substitutions Lemma provides an equality necessary for the cases of the proof of Substitution Lemma 8 that involve a second type substitution. Substitution lemmas for polymorphic languages invariably need a Commuting Substitutions Lemma, but I have not seen it explicitly stated elsewhere, so I do not know a conventional name for it. We need the Useless Substitutions Lemma only because we reuse variables as heap locations. Because the heap does not have free type variables, type substitution does not change the Γ and Υ that describe it. Finally, the weakening lemmas are conventional devices used to argue that unchanged

constructs (e.g., e_1 when (e_0, e_1) becomes (e_0', e_1)) have the same properties in extended contexts (e.g., in the context of a larger heap).

Lemma A.1 (Context Weakening).

1. *If* $\Delta \vdash_{\mathrm{k}} \tau : \kappa$, *then* $\Delta\Delta' \vdash_{\mathrm{k}} \tau : \kappa$.
2. *If* $\vdash_{\mathrm{wf}} \Upsilon$, *then* $\Delta \vdash_{\mathrm{k}} \Upsilon(xp) : \mathtt{A}$.

Proof:

1. The proof is by induction on the derivation of $\Delta \vdash_{\mathrm{k}} \tau : \kappa$.
2. The proof is by induction on the derivation of $\vdash_{\mathrm{wf}} \Upsilon$, using the previous lemma.

Lemma A.2 (Term Weakening). *Suppose* $\vdash_{\mathrm{wf}} \Delta; \Upsilon\Upsilon'; \Gamma\Gamma'$.

1. *If* $\Upsilon; xp \vdash \mathrm{gettype}(\tau, p', \tau')$, *then* $\Upsilon\Upsilon'; xp \vdash \mathrm{gettype}(\tau, p', \tau')$.
2. *If* $\Delta; \Upsilon; \Gamma \vdash_{\mathrm{ltyp}} e : \tau$, *then* $\Delta; \Upsilon\Upsilon'; \Gamma\Gamma' \vdash_{\mathrm{ltyp}} e : \tau$.
3. *If* $\Delta; \Upsilon; \Gamma \vdash_{\mathrm{rtyp}} e : \tau$, *then* $\Delta; \Upsilon\Upsilon'; \Gamma\Gamma' \vdash_{\mathrm{rtyp}} e : \tau$.
4. *If* $\Delta; \Upsilon; \Gamma; \tau \vdash_{\mathrm{styp}} s$, *then* $\Delta; \Upsilon\Upsilon'; \Gamma\Gamma'; \tau \vdash_{\mathrm{styp}} s$.

Proof: The first proof is by induction on the assumed gettype derivation. It follows because if $xp \in \mathrm{Dom}(\Upsilon)$, then $(\Upsilon\Upsilon')(xp) = \Upsilon(xp)$. The other proofs are by simultaneous induction on the assumed typing derivation. Cases SS3.6–8 and SR3.13 can use α-conversion to ensure that $x \notin \mathrm{Dom}(\Gamma\Gamma')$. Cases SL3.1 and SR3.1 follow from the first proof because if $x \in \mathrm{Dom}(\Gamma)$, then $(\Gamma\Gamma')(x) = \Gamma(x)$.

Lemma A.3 (Heap Weakening).

1. *If* $\vdash_{\mathrm{wf}} \cdot; \Upsilon\Upsilon'; \Gamma\Gamma'$ *and* $\Upsilon; \Gamma \vdash_{\mathrm{htyp}} H : \Gamma''$, *then* $\Upsilon\Upsilon'; \Gamma\Gamma' \vdash_{\mathrm{htyp}} H : \Gamma''$.
2. *If* $H \vdash_{\mathrm{refp}} \Upsilon$, *then* $HH' \vdash_{\mathrm{refp}} \Upsilon$.

Proof: The first proof is by induction on the heap-typing derivation, using Term Weakening Lemma 3. The second proof is by induction on the assumed derivation, using the fact that if $x \in \mathrm{Dom}(H)$, then $(HH')(x) = H(x)$.

Lemma A.4 (Useless Substitutions). *Suppose* $\alpha \notin \mathrm{Dom}(\Delta)$.

1. *If* $\Delta \vdash_{\mathrm{k}} \tau' : \kappa$, *then* $\tau'[\tau/\alpha] = \tau'$.
2. *If* $\Delta \vdash_{\mathrm{wf}} \Gamma$, *then* $\Gamma[\tau/\alpha] = \Gamma$.

3. *If* $\vdash_{\text{wf}} \Upsilon$, *then* $(\Upsilon(xp))[\tau/\alpha] = \Upsilon(xp)$.

Proof: The first proof is by induction on the assumed derivation. The other proofs are by induction on the assumed derivation, using the first lemma.

Lemma A.5 (Commuting Substitutions). *If* β *is not free in* τ_2, *then* $\tau_0[\tau_1/\beta][\tau_2/\alpha] = \tau_0[\tau_2/\alpha][\tau_1[\tau_2/\alpha]/\beta]$.

Proof: The proof is by induction on the structure of τ_0. If $\tau_0 = \alpha$, then the result of both substitutions is τ_2, using the assumption that β is not free in τ_2. If $\tau_0 = \beta$, then the result of both substitutions is $\tau_1[\tau_2/\alpha]$. If τ_0 is some other type variable or int, both substitutions are useless. All other cases follow by induction and the definition of substitution.

Lemma A.6 (Substitution). *Suppose* $\Delta \vdash_{\text{ak}} \tau : \kappa$.

1. *If* $\Delta, \alpha{:}\kappa \vdash_{\text{k}} \tau' : \kappa'$, *then* $\Delta \vdash_{\text{k}} \tau'[\tau/\alpha] : \kappa'$.
2. *If* $\Delta, \alpha{:}\kappa \vdash_{\text{ak}} \tau' : \kappa'$, *then* $\Delta \vdash_{\text{ak}} \tau'[\tau/\alpha] : \kappa'$.
3. *If* $\Delta, \alpha{:}\kappa \vdash_{\text{asgn}} \tau'$, *then* $\Delta \vdash_{\text{asgn}} \tau'[\tau/\alpha]$.
4. *If* $\Delta, \alpha{:}\kappa \vdash_{\text{wf}} \Gamma$, *then* $\Delta \vdash_{\text{wf}} \Gamma[\tau/\alpha]$.
5. *If* $\vdash_{\text{wf}} \Delta, \alpha{:}\kappa; \Upsilon; \Gamma$, *then* $\vdash_{\text{wf}} \Delta; \Upsilon; \Gamma[\tau/\alpha]$.
6. *If* $\vdash_{\text{ret}} s$, *then* $\vdash_{\text{ret}} s[\tau/\alpha]$.
7. *If* $\vdash_{\text{wf}} \Upsilon$ *and* $\Upsilon; xp \vdash \text{gettype}(\tau_1, p', \tau_2)$, *then* $\Upsilon; xp \vdash \text{gettype}(\tau_1[\tau/\alpha], p', \tau_2[\tau/\alpha])$.
8. *If* $\Delta, \alpha{:}\kappa; \Upsilon; \Gamma \vdash_{\text{ltyp}} e : \tau'$, *then* $\Delta; \Upsilon; \Gamma[\tau/\alpha] \vdash_{\text{ltyp}} e[\tau/\alpha] : \tau'[\tau/\alpha]$.
   If $\Delta, \alpha{:}\kappa; \Upsilon; \Gamma \vdash_{\text{rtyp}} e : \tau'$, *then* $\Delta; \Upsilon; \Gamma[\tau/\alpha] \vdash_{\text{rtyp}} e[\tau/\alpha] : \tau'[\tau/\alpha]$.
   If $\Delta, \alpha{:}\kappa; \Upsilon; \Gamma; \tau' \vdash_{\text{styp}} s$, *then* $\Delta; \Upsilon; \Gamma[\tau/\alpha] \vdash_{\text{ltyp}} \tau'[\tau/\alpha] : s[\tau/\alpha]$.

Proof:

1. The proof is by induction on the assumed derivation. The nonaxiom cases are by induction. The case for $\tau' = \text{int}$ is trivial. The case where τ' is a type variable is trivial unless $\tau' = \alpha$. In that case, $\Delta(\alpha) = \texttt{B}$, so inverting $\Delta \vdash_{\text{ak}} \tau : \kappa$ ensures $\Delta \vdash_{\text{k}} \tau : \texttt{B}$, as desired. Similarly, the case where τ' has the form $\beta*$ is trivial unless $\beta = \alpha$. In that case, if τ is some type variable α' where $\Delta(\alpha') = \texttt{A}$, then we can derive $\Delta, \alpha'{:}\texttt{A} \vdash_{\text{k}} \alpha'* : \texttt{B}$ as desired. Else inverting $\Delta \vdash_{\text{ak}} \tau : \kappa$ ensures $\Delta \vdash_{\text{k}} \tau : \kappa$, so we can derive $\Delta \vdash_{\text{k}} \tau* : \texttt{B}$ (using the introduction rule for pointer types and possibly the subsumption rule).

2. The proof is by cases on the assumed derivation, using the previous lemma.

3. The proof is by induction on the assumed derivation. The nonaxiom cases are by induction. The cases for int and pointer types are trivial. The case where τ' is a type variable is trivial unless $\tau' = \alpha$. In that case, $\Delta(\alpha) = \mathtt{B}$, so inverting $\Delta \vdash_{\mathrm{ak}} \tau : \kappa$ ensures $\Delta \vdash_{\mathrm{k}} \tau : \mathtt{B}$, as desired.

4. The proof is by induction on the assumed derivation, using Substitution Lemma 1.

5. The lemma is a corollary to the previous lemma.

6. The proof is by induction on the assumed derivation. Type substitution is irrelevant to $\vdash_{\mathrm{ret}}$.

7. The proof is by induction on the assumed derivation. The case where $p' = \cdot$ is trivial. The cases where p' starts with 0 or 1 are by induction. In the remaining case, we have a derivation of the form:

$$\frac{\Upsilon; xp\mathtt{u} \vdash \mathrm{gettype}(\tau_0[\Upsilon(xp)/\beta], p'', \tau_2)}{\Upsilon; xp \vdash \mathrm{gettype}(\exists^{\&}\beta{:}\kappa'.\tau_0, \mathtt{u}p'', \tau_2)}$$

   So by induction, $\Upsilon; xp\mathtt{u} \vdash \mathrm{gettype}(\tau_0[\Upsilon(xp)/\beta][\tau/\alpha], p'', \tau_2[\tau/\alpha])$. So the Commuting Substitutions Lemma ensures
   $\Upsilon; xp\mathtt{u} \vdash \mathrm{gettype}(\tau_0[\tau/\alpha][\Upsilon(xp)[\tau/\alpha]/\beta], p'', \tau_2[\tau/\alpha])$. Useless Substitution Lemma 3 ensures $\Upsilon(xp)[\tau/\alpha] = \Upsilon(xp)$, so
   $\Upsilon; xp\mathtt{u} \vdash \mathrm{gettype}(\tau_0[\tau/\alpha][\Upsilon(xp)/\beta], p'', \tau_2[\tau/\alpha])$, from which we can derive $\Upsilon; xp \vdash \mathrm{gettype}(\exists^{\&}\beta{:}\kappa'.\tau_0[\tau/\alpha], p'', \tau_2[\tau/\alpha])$, as desired.

   Note that this lemma is somewhat unnecessary because a program state reached from a source program that had no nonempty paths can type-check without using the gettype judgment on open types. Put another way, rules SL3.1 and SR3.1 could require that $\Gamma(x)$ is closed if p is nonempty. Rather than prove that restricted type system is sound, I have found it easier just to include this lemma.

8. The proof is by simultaneous induction on the assumed derivations, proceeding by cases on the last rule in the derivation. In each case, we satisfy the hypotheses of the rule after substitution and then use the rule to derive the desired result. So for most cases, we explain just how to conclude the necessary hypotheses. We omit cases SL3.1–4 because they are identical to cases SR3.1–4.

- SR3.1: The left, middle, and right hypotheses follows from Substitution Lemmas 7, 1, and 5, respectively.
- SR3.2: The left hypothesis follows from induction. The right hypothesis follows from Substitution Lemma 1.
- SR3.3: The hypothesis follows from induction.
- SR3.4: The hypothesis follows from induction.
- SR3.5: The hypothesis follows from Substitution Lemma 5.
- SR3.6: The hypothesis follows from induction.
- SR3.7: The hypotheses follow from induction.
- SR3.8: The left and middle hypothesis follow from induction. The right hypothesis follows from Substitution Lemma 3.
- SR3.9: The hypotheses follow from induction.
- SR3.10: The left hypothesis follows from induction. The right hypothesis follows from Substitution Lemma 6.
- SR3.11: We have a derivation of the form:

$$\frac{\Delta, \alpha{:}\kappa; \Upsilon; \Gamma \vdash_{\text{rtyp}} e : \forall \beta{:}\kappa'.\tau_1 \quad \Delta, \alpha{:}\kappa \vdash_{\text{ak}} \tau_0 : \kappa'}{\Delta, \alpha{:}\kappa; \Upsilon; \Gamma \vdash_{\text{rtyp}} e[\tau_0] : \tau_1[\tau_0/\beta]}$$

  The left hypothesis and induction provide $\Delta; \Upsilon; \Gamma \vdash_{\text{rtyp}} e : \forall \beta{:}\kappa'.\tau_1[\tau/\alpha]$. The right hypothesis and Substitution Lemma 2 provide $\Delta \vdash_{\text{ak}} \tau_0[\tau/\alpha] : \kappa'$. So we can derive $\Delta; \Upsilon; \Gamma \vdash_{\text{rtyp}} e[\tau/\alpha][\tau_0[\tau/\alpha]] : \tau_1[\tau/\alpha][\tau_0[\tau/\alpha]/\beta]$. The Commuting Substitutions Lemma ensures the type is what we want.

- SR3.12: We have a derivation of the form:

$$\frac{\Delta, \alpha{:}\kappa; \Upsilon; \Gamma \vdash_{\text{rtyp}} e : \tau_1[\tau_0/\beta] \quad \Delta, \alpha{:}\kappa \vdash_{\text{ak}} \tau_0 : \kappa' \quad \Delta, \alpha{:}\kappa \vdash_{\text{k}} \exists^{\phi} \beta{:}\kappa'.\tau_1 : \mathtt{A}}{\Delta, \alpha{:}\kappa; \Upsilon; \Gamma \vdash_{\text{rtyp}} \mathsf{pack}\ \tau_0, e\ \mathsf{as}\ \exists^{\phi} \beta{:}\kappa'.\tau_1 : \exists^{\phi} \beta{:}\kappa'.\tau_1}$$

  The left hypothesis and induction provide $\Delta; \Upsilon; \Gamma[\tau/\alpha] \vdash_{\text{rtyp}} e[\tau/\alpha] : \tau_1[\tau_0/\beta][\tau/\alpha]$, which by the Commuting Substitutions Lemma ensures $\Delta; \Upsilon; \Gamma[\tau/\alpha] \vdash_{\text{rtyp}} e[\tau/\alpha] : \tau_1[\tau/\alpha][\tau_0[\tau/\alpha]/\beta]$. The middle hypothesis and Substitution Lemma 2 provide $\Delta \vdash_{\text{ak}} \tau_0[\tau/\alpha] : \kappa'$. The right hypothesis and Substitution Lemma 1 provide $\Delta \vdash_{\text{k}} \exists^{\phi} \beta{:}\kappa'.\tau_1[\tau/\alpha] : \mathtt{A}$. So we can derive $\Delta; \Upsilon; \Gamma \vdash_{\text{rtyp}} \mathsf{pack}\ \tau_0[\tau/\alpha], e[\tau/\alpha]\ \mathsf{as}\ \exists^{\phi} \beta{:}\kappa'.\tau_1[\tau/\alpha] : \exists^{\phi} \beta{:}\kappa'.\tau_1[\tau/\alpha]$, as desired.

- SR3.13: The left hypothesis follows from induction. The right hypothesis follows from Substitution Lemma 6.

- SR3.14: The left hypothesis follows from induction (using implicit context reordering). The well-formedness hypothesis follows from Substitution Lemma 5.
- SS3.1–6: In each case, all hypotheses follow from induction.
- SS3.7–8: In both cases, Substitution Lemma 1 provides the kinding hypothesis and induction (and context reordering) provides the typing hypotheses.

Lemma A.7 (Typing Well-Formedness).

1. *If* $\vdash_{\text{wf}} \Upsilon$, $\Upsilon; xp \vdash \text{gettype}(\tau, p', \tau')$, *and* $\Delta \vdash_{\text{k}} \tau : \mathtt{A}$, *then* $\Delta \vdash_{\text{k}} \tau' : \mathtt{A}$.
2. *If* $\Delta; \Upsilon; \Gamma \vdash_{\text{ltyp}} e : \tau$, *then* $\vdash_{\text{wf}} \Delta; \Upsilon; \Gamma$ *and* $\Delta \vdash_{\text{k}} \tau : \mathtt{A}$.
3. *If* $\Delta; \Upsilon; \Gamma \vdash_{\text{rtyp}} e : \tau$, *then* $\vdash_{\text{wf}} \Delta; \Upsilon; \Gamma$ *and* $\Delta \vdash_{\text{k}} \tau : \mathtt{A}$.
4. *If* $\Delta; \Upsilon; \Gamma; \tau \vdash_{\text{styp}} s$, *then* $\vdash_{\text{wf}} \Delta; \Upsilon; \Gamma$.
   If $\Delta; \Upsilon; \Gamma; \tau \vdash_{\text{styp}} s$ *and* $\vdash_{\text{ret}} s$, *then* $\Delta \vdash_{\text{k}} \tau : \mathtt{A}$.

Proof: The first proof is by induction on the gettype derivation. The case where $p' = \cdot$ is trivial. The cases where p' starts with 0 or 1 are by induction and inversion of the kinding derivation. In the remaining case, the induction hypothesis applies by inverting the kinding derivation (to get $\Delta, \alpha{:}\kappa \vdash_{\text{k}} \tau_0 : \mathtt{A}$ where $\tau = \exists^{\&}\alpha{:}\kappa.\tau_0$), inverting the gettype derivation (to ensure $xp \in \text{Dom}\Upsilon$, so $\vdash_{\text{wf}} \Upsilon$ provides $\cdot \vdash_{\text{k}} \Upsilon(xp) : \mathtt{A}$), Context Weakening Lemma 2 (to get $\Delta \vdash_{\text{k}} \Upsilon(xp) : \mathtt{A}$), and Substitution Lemma 1 (to get $\Delta \vdash_{\text{k}} \tau_0[\Upsilon(xp)/\alpha] : \mathtt{A}$).

The remaining proofs are by simultaneous induction on the assumed typing derivations. Most cases follow trivially from an explicit hypothesis or from induction and the definition of $\Delta \vdash_{\text{k}} \tau : \mathtt{A}$. Cases SL3.1 and SR3.1 use the first lemma. Case SR3.11 uses Substitution Lemma 1. Case SR3.13 uses the definition of $\vdash_{\text{wf}} \Delta; \Upsilon; \Gamma$ to determine the function-argument type has kind $\mathtt{A}$. The statement cases must argue about whether the contained expressions must return. As examples, case SS3.1 uses the fact that $\not\vdash_{\text{ret}} e$ to vacuously satisfy the conclusion for τ, and case SS3.3 uses the fact that if $\vdash_{\text{ret}} s_1; s_2$, then one of the invocations of the induction hypothesis provide $\Delta \vdash_{\text{k}} \tau : \mathtt{A}$.

Lemma A.8 (Return Preservation). *If* $\vdash_{\text{ret}} s$ *and* $H; s \xrightarrow{\text{s}} H'; s'$, *then* $\vdash_{\text{ret}} s'$.

Proof: The proof is by induction on the derivation of $H; s \xrightarrow{\text{s}} H'; s'$, proceeding by cases on the last rule in the derivation:

- DS3.1: $s = \mathsf{let}\ x = v;\ s'$ and $\vdash_{\text{ret}} s$ implies $\vdash_{\text{ret}} s'$.

- DS3.2: $s = (v; s')$ and $\vdash_{\text{ret}} s$ implies $\vdash_{\text{ret}} s'$ (because $\not\vdash_{\text{ret}} v$).
- DS3.3: $s' = \mathsf{return}\ v$, so trivially $\vdash_{\text{ret}} s'$.
- DS3.4: $s = \mathsf{if}\ v\ s_1\ s_2$ and $\vdash_{\text{ret}} s$ implies $\vdash_{\text{ret}} s_1$ and $\vdash_{\text{ret}} s_2$, so in both cases $\vdash_{\text{ret}} s'$.
- DS3.5: The argument for the previous case applies.
- DS3.6: This case holds vacuously because $\not\vdash_{\text{ret}} s$.
- DS3.7: $s = \mathsf{open}\ v\ \mathsf{as}\ \alpha, x;\ s''$ and $\vdash_{\text{ret}} s$ implies $\vdash_{\text{ret}} s''$. Substitution Lemma 6 ensures $\vdash_{\text{ret}} s''[\tau/\alpha]$ for any τ, so we can derive $\vdash_{\text{ret}} \mathsf{let}\ x = v;\ s''[\tau/\alpha]$.
- DS3.8: This case is analogous to the previous one.
- DS3.9: For each conclusion, if $\vdash_{\text{ret}} s$, then $\vdash_{\text{ret}} s'$ because the form of the subexpression is irrelevant.
- DS3.10: $s = s_1; s_2$, so if $\vdash_{\text{ret}} s$, then either $\vdash_{\text{ret}} s_1$ or $\vdash_{\text{ret}} s_2$. In the former case, the induction hypothesis lets us use one of the composition-introduction rules to derive $\vdash_{\text{ret}} s'$. In the latter case, the other rule applies regardless of the statement that s_1 becomes.
- DS3.11: If $\vdash_{\text{ret}} s$, then $\vdash_{\text{ret}} s'$ because the form of the subexpression is irrelevant.

Lemma A.9 (Canonical Forms). *Suppose* $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v : \tau$.

- *If* $\tau = \text{int}$, *then* $v = i$ *for some* i.
- *If* $\tau = \tau_0 \times \tau_1$, *then* $v = (v_0, v_1)$ *for some* v_0 *and* v_1.
- *If* $\tau = \tau_0 \to \tau_1$, *then* $v = (\tau_0\ x) \to \tau_1\ s$ *for some* x *and* s.
- *If* $\tau = \tau'*$, *then* $v = \&xp$ *for some* x *and* p.
- *If* $\tau = \forall\alpha{:}\kappa.\tau'$, *then* $v = \Lambda\alpha{:}\kappa.f$ *for some* f.
- *If* $\tau = \exists^{\delta}\alpha{:}\kappa.\tau'$, *then* $v = \mathsf{pack}\ \tau'', v'\ \mathsf{as}\ \exists^{\delta}\alpha{:}\kappa.\tau'$ *for some* τ'' *and* v'.
- *If* $\tau = \exists^{\&}\alpha{:}\kappa.\tau'$, *then* $v = \mathsf{pack}\ \tau'', v'\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'$ *for some* τ'' *and* v'.

Proof: The proof is by inspection of the rules for $\vdash_{\text{rtyp}}$ and the form of values.

Lemma A.10 (Path Extension).

1. *Suppose* $\mathrm{get}(v, p, v')$.
   If $v' = (v_0, v_1)$, *then* $\mathrm{get}(v, p0, v_0)$ *and* $\mathrm{get}(v, p1, v_1)$, *else we cannot derive* $\mathrm{get}(v, pip', v'')$ *for any* i, p', *and* v''.
   If $v' = \mathsf{pack}\ \tau', v_0\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau$, *then* $\mathrm{get}(v, p\mathtt{u}, v_0)$, *else we cannot derive* $\mathrm{get}(v, p\mathtt{u}p', v'')$ *for any* p' *and* v''.

2. *Suppose* $\Upsilon; xp \vdash \mathrm{gettype}(\tau, p', \tau')$.
   If $\tau' = \tau_0 \times \tau_1$, *then* $\Upsilon; xp \vdash \mathrm{gettype}(\tau, p'0, \tau_0)$ *and* $\Upsilon; xp \vdash \mathrm{gettype}(\tau, p'1, \tau_1)$.
   If $\tau' = \exists^{\&}\alpha{:}\kappa.\tau_0$ *and* $xp \in \mathrm{Dom}(\Upsilon)$, *then* $\Upsilon; xp \vdash \mathrm{gettype}(\tau, p'\mathtt{u}, \tau_0[\Upsilon(xp)/\alpha])$.

Proof:

1. The proof is by induction on the length of p. If $p = \cdot$, then $v = v'$ and the result follows from inspection of the get relation (because $\cdot p_1 = p_1$ for all p_1). For longer p, we proceed by cases on the leftmost element of p. In each case, inverting the $\mathrm{get}(v, p, v')$ derivation and the induction hypothesis suffice.

2. The proof is by induction on the length of p'. If $p' = \cdot$, then $\tau = \tau'$ and the result follows from inspection of the gettype relation (because $\cdot p_1 = p_1$ for all p_1). For longer p', we proceed by cases on the leftmost element of p'. In each case, inverting the $\Upsilon; xp \vdash \mathrm{gettype}(\tau, p', \tau')$ derivation and the induction hypothesis suffice.

Lemma A.11 (Heap-Object Safety).

1. *There is at most one* v_2 *such that* $\mathrm{get}(v_1, p, v_2)$.

2. *If* $\mathrm{get}(v_0, p_1, v_1)$ *and* $\mathrm{get}(v_0, p_1p_2, v_2)$, *then* $\mathrm{get}(v_1, p_2, v_2)$.

3. *Suppose*

   - $H \vdash_{\mathrm{refp}} \Upsilon$ *and* $\Upsilon; \Gamma \vdash_{\mathrm{htyp}} H : \Gamma$
   - $\mathrm{get}(H(x), p_1, v_1)$ *and* $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{rtyp}} v_1 : \tau_1$
   - $\Upsilon; xp_1 \vdash \mathrm{gettype}(\tau_1, p_2, \tau_2)$

   Then:

   - *There exists a* v_2 *such that* $\mathrm{get}(H(x), p_1p_2, v_2)$. *Also,* $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{rtyp}} v_2 : \tau_2$.
   - *For all* v_2', *there exists a* v_1' *such that* $\mathrm{set}(v_1, p_2, v_2', v_1')$.

   Corollary: If $H \vdash_{\mathrm{refp}} \Upsilon$, $U; \Gamma \vdash_{\mathrm{htyp}} H : \Gamma$, *and* $\Upsilon; x\cdot \vdash \mathrm{gettype}(\tau_1, p_2, \tau_2)$, *then the conclusions hold with* $p_1 = \cdot$ *and* $v_1 = H(x)$.

4. *Suppose in addition to the previous lemma's assumptions,* $\cdot \vdash_{\text{asgn}} \tau_2$. *Then for all* p', $xp_1p_2p' \notin \text{Dom}(\Upsilon)$.

5. *Suppose in addition to the previous lemma's assumptions,* $\text{set}(v_1, p_2, v_2', v_1')$ *and* $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_2' : \tau_2$. *Then* $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_1' : \tau_1$ *and if* $xp_1p' \in \text{Dom}(\Upsilon)$, *there are* v'', τ'', α, *and* κ *such that* $\text{get}(v_1', p', \mathsf{pack}\ \Upsilon(xp_1p'), v''\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'')$. *Corollary: If* $H \vdash_{\text{refp}} \Upsilon$, $\Upsilon; \Gamma \vdash_{\text{htyp}} H : \Gamma$, $\Upsilon; x\cdot \vdash \text{gettype}(\tau_1, p_2, \tau_2)$, $\cdot \vdash_{\text{asgn}} \tau_2$, $\text{set}(H(x), p_2, v_2', v_1')$, *and* $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_2' : \tau_2$ *then the conclusions hold with* $p_1 = \cdot$ *and* $v_1 = H(x)$.

Proof:

1. The proof is by induction on the length of p.

2. The proof is by induction on the length of p_1.

3. The proof is by induction on the length of p_2. If $p_2 = \cdot$, the gettype relation ensures $\tau_1 = \tau_2$ and the get relation ensures $\text{get}(H(x), p_1\cdot, v_1)$. So letting $v_2 = v_1$, the assumption $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_1 : \tau_1$ means $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_2 : \tau_2$. We can trivially derive $\text{set}(v_1, \cdot, v_2', v_2')$. For longer paths, we proceed by cases on the leftmost element:

   - $p_2 = 0p_3$:

     Inverting the assumption $\Upsilon; xp_1 \vdash \text{gettype}(\tau_1, 0p_3, \tau_2)$ provides $\underline{\Upsilon; xp_10 \vdash \text{gettype}(\tau_{10}, p_3, \tau_2)}$ where $\tau_1 = \tau_{10} \times \tau_{11}$. Inverting the assumption $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_1 : \tau_{10} \times \tau_{11}$ provides $v_1 = (v_{10}, v_{11})$ and $\underline{\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_{10} : \tau_{10}}$. Applying Path Extension Lemma 1 to the assumption $\text{get}(H(x), p_1, (v_{10}, v_{11}))$ provides $\underline{\text{get}(H(x), p_10, v_{10})}$. So the induction hypothesis applies to the underlined results, using p_10 for p_1, v_{10} for v_1, τ_{10} for τ_1, p_3 for p_2, and τ_2 for τ_2.

     Therefore, there exists a v_2 such that $\text{get}(H(x), p_10p_3, v_2)$ and $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_2 : \tau_2$, as desired. Furthermore, for all v_2' there exists a v_{10}' such that $\text{set}(v_{10}, p_3, v_2', v_{10}')$. Hence we can derive $\text{set}((v_{10}, v_{11}), 0p_3, v_2', (v_{10}', v_{11}))$, which satisfies the desired result (letting $v_1' = (v_{10}', v_{11})$).

   - $p_2 = 1p_3$: This case is analogous to the previous one.

   - $p_2 = \mathtt{u}p_3$:

     Inverting the assumption $\Upsilon; xp_1 \vdash \text{gettype}(\tau_1, \mathtt{u}p_3, \tau_2)$ provides $\underline{\Upsilon; xp_1\mathtt{u} \vdash \text{gettype}(\tau_3[\Upsilon(xp_1)/\alpha], p_3, \tau_2)}$ where $\tau_1 = \exists^{\&}\alpha{:}\kappa.\tau_3$. Inverting the assumption $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_1 : \exists^{\&}\alpha{:}\kappa.\tau_3$ provides $\underline{\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_3 : \tau_3[\tau_4/\alpha]}$

where $v_1 = \mathsf{pack}\ \tau_4, v_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3$. Applying Path Extension Lemma 1 to the assumption $\mathrm{get}(H(x), p_1, \mathsf{pack}\ \tau_4, v_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3)$ provides $\underline{\mathrm{get}(H(x), p_1\mathtt{u}, v_3)}$. From $\mathrm{get}(H(x), p_1, \mathsf{pack}\ \tau_4, v_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3)$, Heap-Object Safety Lemma 1, and $H \vdash_{\mathrm{refp}} \Upsilon$, we know $\tau_4 = \Upsilon(xp_1)$. So the induction hypothesis applies to the underlined results, using $p_1\mathtt{u}$ for p_1, v_3 for v_1, $\tau_3[\Upsilon(xp_1)/\alpha]$ for τ_1, p_3 for p_2, and τ_2 for τ_2.

Therefore, there exists a v_2 such that $\mathrm{get}(H(x), p_1\mathtt{u}p_3, v_2)$ and $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{rtyp}} v_2 : \tau_2$, as desired. Furthermore, for all v_2' there exists a v_3' such that $\mathrm{set}(v_3, p_3, v_2', v_3')$. Hence we can derive $\mathrm{set}(\mathsf{pack}\ \tau_4, v_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3, \mathtt{u}p_3, v_2', \mathsf{pack}\ \tau_4, v_3'\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3)$, which satisfies the desired result (letting $v_1' = \mathsf{pack}\ \tau_4, v_3'\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3$).

The corollary holds because $\mathrm{get}(H(x), \cdot, H(x))$ and $\Upsilon; \Gamma \vdash_{\mathrm{htyp}} H : \Gamma$ ensures $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{rtyp}} H(x) : \tau_1$.

4. Heap-Object Safety Lemmas 1 and 3 ensure there is exactly one v_2 such that $\mathrm{get}(H(x), p_1p_2, v_2)$. Furthermore, $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{rtyp}} v_2 : \tau_2$. The proof proceeds by induction on the structure of τ_2.

   If $\tau_2 = \mathrm{int}$, the Canonical Forms Lemma ensures $v_2 = i$ for some i. Hence Path Extension Lemma 1 ensures we cannot derive $\mathrm{get}(H(x), p_1p_2p', v'')$ unless $p' = \cdot$ (and therefore $v'' = i$). So $\mathrm{get}(H(x), p_1p_2p', \mathsf{pack}\ \tau_0, v_0\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_0')$ is impossible , but it is necessary for $xp_1p_2p' \in \mathrm{Dom}(\Upsilon)$.

   The cases for $\tau_2 = \tau_3*$, $\tau_2 = \tau_3 \to \tau_4$, $\tau_2 = \exists^{\delta}\alpha{:}\kappa.\tau_3$, and $\tau_2 = \forall\alpha{:}\kappa.\tau_3$ are analogous to the case for int, replacing i with a different form of value.

   If $\tau_2 = \alpha$ or $\tau_2 = \exists^{\&}\alpha{:}\kappa.\tau_3$, the lemma holds vacuously because we cannot derive $\cdot \vdash_{\mathrm{asgn}} \tau_2$.

   If $\tau_2 = \tau_3 \times \tau_4$, the Canonical Forms Lemma ensures $v_2 = (v_3, v_4)$. Hence Path Extension Lemma 1 ensures we can derive $\mathrm{get}(H(x), p_1p_2p', v'')$ only if $p' = \cdot$, $p' = 0p''$, or $p' = 1p''$. If $p' = \cdot$, then $\mathrm{get}(H(x), p_1p_2p', \mathsf{pack}\ \tau_0, v_0\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_0')$ is impossible, but it is necessary for $xp_1p_2p' \in \mathrm{Dom}(\Upsilon)$. If $p' = 0p''$, applying Path Extension Lemma 2 to the assumption $\Upsilon; xp_1 \vdash \mathrm{gettype}(\tau_1, p_2, \tau_2)$ provides $\underline{\Upsilon; xp_1 \vdash \mathrm{gettype}(\tau_1, p_20, \tau_3)}$. Inverting the assumption $\cdot \vdash_{\mathrm{asgn}} \tau_3 \times \tau_4$ provides $\underline{\cdot \vdash_{\mathrm{asgn}} \tau_3}$. With the underlined results and the assumptions $\underline{\mathrm{get}(H(x), p_1, v_1)}$ and $\underline{\cdot; \Upsilon; \Gamma \vdash_{\mathrm{rtyp}} v_1 : \tau_1}$, the induction hypothesis applies (using p_20 for p_2 and τ_3 for τ_2), so $xp_1p_20p'' \notin \mathrm{Dom}(\Upsilon)$, as desired. The argument for $p' = 1p''$ is analogous.

5. The proof is by induction on the length of p_2. If $p_2 = \cdot$, the set relation ensures $v_1' = v_2'$. and the gettype relation ensures $\tau_2 = \tau_1$. Hence the

assumption $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_2' : \tau_2$ means $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_1' : \tau_1$. Heap Lemma 4 ensures $xp_1 \cdot p' \notin \text{Dom}(\Upsilon)$, so the second conclusion holds vacuously. For longer paths, we proceed by cases on the leftmost element:

- $p_2 = 0p_3$: Inverting the assumption $\Upsilon; xp_1 \vdash \text{gettype}(\tau_1, 0p_3, \tau_2)$ provides $\underline{\Upsilon; xp_10 \vdash \text{gettype}(\tau_{10}, p_3, \tau_2)}$ where $\tau_1 = \tau_{10} \times \tau_{11}$. Inverting the assumption $\text{set}(v_1, 0p_3, v_2', v_1')$ provides $\underline{\text{set}(v_{10}, p_3, v_2', v_{10}')}$ where $v_1 = (v_{10}, v_{11})$ and $v_1' = (v_{10}', v_{11})$. Applying Path Extension Lemma 1 to the assumption $\text{get}(H(x), p_1, (v_{10}, v_{11}))$ provides $\underline{\text{get}(H(x), p_10, v_{10})}$. Inverting the assumption $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} (v_{10}, v_{11}) : \tau_{10} \times \tau_{11}$ provides $\underline{\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_{10} : \tau_{10}}$. With the underlined results and the assumptions $\underline{\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_2' : \tau_2}$ and $\underline{\cdot \vdash_{\text{asgn}} \tau_2}$, the induction hypothesis applies (using p_10 for p_1, p_3 for p_2, v_{10} for v_1, τ_{10} for τ_1, τ_2 for τ_2, v_2' for v_2', and v_{10}' for v_1').
  Hence $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_{10}' : \tau_{10}$ and if $xp_10p'' \in \text{Dom}(\Upsilon)$, then $\text{get}(v_{10}', p'', \mathsf{pack}\ \Upsilon(xp_10p''), v''\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'')$. So we can derive $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} (v_{10}', v_{11}) : \tau_{10} \times \tau_{11}$, as desired. If $xp_1p' \in \text{Dom}(\Upsilon)$, then $H \vdash_{\text{refp}} \Upsilon$ provides $\text{get}(H(x), p_1p', \mathsf{pack}\ \Upsilon(xp_1p'), v''\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'')$. Because $\text{get}(H(x), p_1, (v_{10}, v_{11}))$, Path Extension Lemma 1 ensures that p' has the form $\cdot$, $0p''$, or $1p''$. Heap-Object Safety Lemma 1 precludes $p' = \cdot$. If $p' = 0p''$, the induction provides the result we need. If $p' = 1p''$, applying Heap-Object Safety Lemma 2 provides $\text{get}((v_{10}, v_{11}), 1p'', \mathsf{pack}\ \Upsilon(xp_1p'), v''\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'')$, which by inversion provides $\text{get}(v_{11}, p'', \mathsf{pack}\ \Upsilon(xp_1p'), v''\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'')$. So we can derive $\text{get}((v_{10}', v_{11}), 1p'', \mathsf{pack}\ \Upsilon(xp_1p'), v''\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'')$, as desired.
- $p_2 = 1p_3$: This case is analogous to the previous one.
- $p_2 = \mathtt{u}p_3$: Inverting the assumption $\Upsilon; xp_1 \vdash \text{gettype}(\tau_1, \mathtt{u}p_3, \tau_2)$ provides $\underline{\Upsilon; xp_1\mathtt{u} \vdash \text{gettype}(\tau_3[\Upsilon(xp_1)/\alpha], p_3, \tau_2)}$ where $\tau_1 = \exists^{\&}\alpha{:}\kappa.\tau_3$. Inverting the assumption $\text{set}(v_1, \mathtt{u}p_3, v_2', v_1')$ provides $\underline{\text{set}(v_3, p_3, v_2', v_3')}$ where $v_1 = \mathsf{pack}\ \tau_4, v_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3$ and $v_1' = \mathsf{pack}\ \tau_4, v_3'\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3$. Applying Path Extension Lemma 1 to the assumption $\text{get}(H(x), p_1, \mathsf{pack}\ \tau_4, v_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3)$ provides $\underline{\text{get}(H(x), p_1\mathtt{u}, v_3)}$. Inverting the assumption $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} \mathsf{pack}\ \tau_4, v_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3 : \exists^{\&}\alpha{:}\kappa.\tau_3$ provides $\underline{\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_3 : \tau_3[\tau_4/\alpha]}$. From $\text{get}(H(x), p_1, \mathsf{pack}\ \tau_4, v_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3)$, Heap-Object Safety Lemma 1, and $H \vdash_{\text{refp}} \Upsilon$, we know $\tau_4 = \Upsilon(xp_1)$. So with the underlined results and the assumptions $\underline{\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_2' : \tau_2}$ and $\underline{\cdot \vdash_{\text{asgn}} \tau_2}$, the induction hypothesis applies (using $p_1\mathtt{u}$ for p_1, p_3 for p_2, v_3 for v_1, $\tau_3[\Upsilon(xp_1)/\alpha]$ for τ_1, τ_2 for τ_2, v_2' for v_2', and v_3' for v_1').

Hence $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_3' : \tau_3[\Upsilon(xp_1)/\alpha]$ and if $xp_1\mathtt{u}p'' \in \mathrm{Dom}(\Upsilon)$, then $\mathrm{get}(v_3', p'', \mathsf{pack}\ \Upsilon(xp_1\mathtt{u}p''), v'' \ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'')$. So we can derive $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} \mathsf{pack}\ \tau_4, v_3'\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3 : \exists^{\&}\alpha{:}\kappa.\tau_3$, as desired. If $xp_1p' \in \mathrm{Dom}(\Upsilon)$, then $H \vdash_{\text{refp}} \Upsilon$ provides $\mathrm{get}(H(x), p_1p', \mathsf{pack}\ \Upsilon(xp_1p'), v'' \ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'')$. Because $\mathrm{get}(H(x), p_1, \mathsf{pack}\ \tau_4, v_3\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau_3)$, Path Extension Lemma 1 ensures that p' has the form $\cdot$ or $\mathtt{u}p''$. The case $p' = \cdot$ is trivial because $\mathrm{get}(v_1', \cdot, v_1')$ (the witness type did not change, so it is okay if $xp_1 \in \mathrm{Dom}(\Upsilon)$). The case $\mathtt{u}p''$ follows from induction.

The corollary holds because $\mathrm{get}(H(x), \cdot, H(x))$ and $\Upsilon; \Gamma \vdash_{\text{htyp}} H : \Gamma$ ensures $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} H(x) : \tau_1$.

Definition A.12 (Extension). Γ_2 *(or* Υ_2*)* <u>*extends*</u> Γ_1 *(or* Υ_1*) if there exists a* Γ_3 *(or* Υ_3*) such that* $\Gamma_2 = \Gamma_1\Gamma_3$ *(or* $\Upsilon_2 = \Upsilon_1\Upsilon_3$*).*

Lemma A.13 (Term Preservation). *Suppose* $\Upsilon; \Gamma \vdash_{\text{htyp}} H : \Gamma$ *and* $H \vdash_{\text{refp}} \Upsilon$.

- *If* $\cdot; \Upsilon; \Gamma \vdash_{\text{ltyp}} e : \tau$ *and* $H; e \xrightarrow{\text{l}} H'; e'$*, then there exist* Γ' *and* Υ' *extending* Γ *and* Υ *such that* $\Upsilon'; \Gamma' \vdash_{\text{htyp}} H' : \Gamma'$*,* $H' \vdash_{\text{refp}} \Upsilon'$*, and* $\cdot; \Upsilon'; \Gamma' \vdash_{\text{ltyp}} e' : \tau$.

- *If* $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} e : \tau$ *and* $H; e \xrightarrow{\text{r}} H'; e'$*, then there exist* Γ' *and* Υ' *extending* Γ *and* Υ *such that* $\Upsilon'; \Gamma' \vdash_{\text{htyp}} H' : \Gamma'$*,* $H' \vdash_{\text{refp}} \Upsilon'$*, and* $\cdot; \Upsilon'; \Gamma' \vdash_{\text{rtyp}} e' : \tau$.

- *If* $\cdot; \Upsilon; \Gamma; \tau \vdash_{\text{styp}} s$ *and* $H; s \xrightarrow{\text{s}} H'; s'$*, then there exist* Γ' *and* Υ' *extending* Γ *and* Υ *such that* $\Upsilon'; \Gamma' \vdash_{\text{htyp}} H' : \Gamma'$*,* $H' \vdash_{\text{refp}} \Upsilon'$*, and* $\cdot; \Upsilon'; \Gamma'; \tau \vdash_{\text{styp}} s'$.

Proof: The proof is by simultaneous induction on the assumed derivations that the term can take a step, proceeding by cases on the last rule used. Except where noted, we use $H' = H$, $\Gamma' = \Gamma$, and $\Upsilon' = \Upsilon$.

- DL3.1: Inverting $\cdot; \Upsilon; \Gamma \vdash_{\text{ltyp}} xp.i : \tau$ provides $\Upsilon; x\cdot \vdash \mathrm{gettype}(\Gamma(x), p, \tau_0 \times \tau_1)$ (where $\tau = \tau_i$), $\cdot \vdash_{\text{k}} \Gamma(x) : \mathtt{A}$, and $\vdash_{\text{wf}} \cdot; \Upsilon; \Gamma$. Applying Path Extension Lemma 2 provides $\Upsilon; x\cdot \vdash \mathrm{gettype}(\Gamma(x), pi, \tau_i)$, so we can derive $\cdot; \Upsilon; \Gamma \vdash_{\text{ltyp}} xpi : \tau_i$.

- DL3.2: Inverting $\cdot; \Upsilon; \Gamma \vdash_{\text{ltyp}} *\&xp : \tau$ provides $\cdot; \Upsilon; \Gamma \vdash_{\text{ltyp}} xp : \tau$.

- DL3.3: Inverting $\cdot; \Upsilon; \Gamma \vdash_{\text{ltyp}} *e_1 : \tau$ provides $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} e_1 : \tau*$ and $\cdot \vdash_{\text{k}} e_1 : \mathtt{A}$. So the induction hypothesis applies to $H; e_1 \xrightarrow{\text{r}} H'; e_1'$. Using the induction, we can derive $\cdot; \Upsilon'; \Gamma' \vdash_{\text{ltyp}} *e_1' : \tau$.

- DL3.4: Inverting $\cdot; \Upsilon; \Gamma \vdash_{\text{ltyp}} e_1.i : \tau$ provides $\cdot; \Upsilon; \Gamma \vdash_{\text{ltyp}} e_1 : \tau_0 \times \tau_1$ (where $\tau = \tau_i$) . So the induction hypothesis applies to $H; e_1 \xrightarrow{\text{l}} H'; e_1'$. Using the induction, we can derive $\cdot; \Upsilon'; \Gamma' \vdash_{\text{ltyp}} e_1'.i : \tau$.

- DR3.1: Inverting $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} xp : \tau$ provides $\Upsilon; x\cdot \vdash \text{gettype}(\Gamma(x), p, \tau)$. So Heap-Object Safety Lemmas 1 and 3 provide $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v : \tau$.

- DR3.2: Inverting $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} xp{=}v : \tau$ provides $\Upsilon; x\cdot \vdash \text{gettype}(\Gamma(x), p, \tau)$, $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v : \tau$, and $\cdot \vdash_{\text{asgn}} \tau$. Heap-Object Safety Lemma 5 provides $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v' : \Gamma(x)$ and all $xp' \in \text{Dom}(\Upsilon)$ are still correct in the sense of $H \vdash_{\text{refp}} \Upsilon$. So letting $H' = H, x \mapsto v'$, $\Gamma' = \Gamma$, and $\Upsilon' = \Upsilon$, we can derive the needed results.

- DR3.3: Inverting $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} *\&xp : \tau$ provides $\cdot; \Upsilon; \Gamma \vdash_{\text{ltyp}} xp : \tau$, which implies $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} xp : \tau$ (because SL3.1 and SR3.1 have identical hypotheses).

- DR3.4: Inverting $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} (v_0, v_1).i : \tau_i$ provides $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v_i : \tau_i$.

- DR3.5: Inverting $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} ((\tau_1\ x) \to \tau\ s)(v) : \tau$ provides $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v : \tau_1$, $\vdash_{\text{ret}} s$, and $\cdot; \Upsilon; \Gamma, x{:}\tau_1 \vdash_{\text{rtyp}} \tau : s$. Using SS3.6 and SR3.10, these results let us derive $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} \mathsf{call}\ (\mathsf{let}\ x = v;\ s) : \tau$.

- DR3.6: Inverting $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} \mathsf{call\ return}\ v : \tau$ provides $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v : \tau$.

- DR3.7: Inverting $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} (\Lambda\alpha{:}\kappa.f)[\tau_1] : \forall\alpha{:}\kappa.\tau_2$ provides $\alpha{:}\kappa; \Upsilon; \Gamma \vdash_{\text{rtyp}} f : \tau_2$ and $\cdot \vdash_{\text{ak}} \tau_1 : \kappa$. So Substitution Lemma 8 provides $\cdot; \Upsilon; \Gamma[\tau_1/\alpha] \vdash_{\text{rtyp}} f[\tau_1/\alpha] : \tau_2[\tau_1/\alpha]$. Because $\vdash_{\text{wf}} \cdot; \Upsilon; \Gamma$, Useless Substitution Lemma 2 ensures $\Gamma[\tau_1/\alpha] = \Gamma$. So $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} f[\tau_1/\alpha] : \tau_2[\tau_1/\alpha]$.

- DR3.8–10: The arguments for each of the conclusions are very similar. Inverting the typing derivation provides that the induction hypothesis applies to the contained s (for DR3.8) or e (for DR3.9 or DR3.10). The induction provides $\Upsilon'; \Gamma' \vdash_{\text{htyp}} H' : \Gamma'$, $H \vdash_{\text{refp}} \Upsilon'$, and the appropriate typing judgment for the transformed contained term. To conclude the appropriate typing judgment for the transformed outer term, we use the same static rule as the original typing derivation. For DR3.8, we also need the Return Preservation Lemma to use SR3.10. For cases with other contained terms (e.g. (v, e)), we use the Term Weakening Lemma to type-check the unchanged terms under Υ' and Γ'. (This argument is why we require Υ' and Γ' to extend Υ and Γ.)

- DS3.1: Inverting $\cdot; \Upsilon; \Gamma; \tau \vdash_{\text{styp}} \mathsf{let}\ x = v;\ s$ provides $\underline{\cdot; \Upsilon; \Gamma, x{:}\tau'; \tau \vdash_{\text{styp}} s}$ and $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v : \tau'$. Let $H' = H, x \mapsto v$, $\Gamma' = \Gamma, x{:}\tau'$, and $\Upsilon' = \Upsilon$. The Typing Well-Formedness Lemma provides $\Delta \vdash_{\text{k}} \tau' : \mathtt{A}$ and $\vdash_{\text{wf}} \cdot; \Upsilon; \Gamma$, so $\vdash_{\text{wf}} \cdot; \Upsilon'; \Gamma'$. So Heap Weakening Lemma 1 provides $\Upsilon'; \Gamma' \vdash_{\text{htyp}} H : \Gamma$, so $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v : \tau'$ provides $\underline{\Upsilon'; \Gamma' \vdash_{\text{htyp}} H' : \Gamma'}$. Heap Weakening Lemma 2 provides $\underline{H' \vdash_{\text{refp}} \Upsilon'}$. The underlined results are our obligations.

- DS3.2: Inverting $\cdot; \Upsilon; \Gamma; \tau \vdash_{\text{styp}} v; s$ provides $\cdot; \Upsilon; \Gamma; \tau \vdash_{\text{styp}} s$.
- DS3.3: Inverting $\cdot; \Upsilon; \Gamma; \tau \vdash_{\text{styp}} \mathsf{return}\ v; s$ provides $\cdot; \Upsilon; \Gamma; \tau \vdash_{\text{styp}} \mathsf{return}\ v$.
- DS3.4: Inverting $\cdot; \Upsilon; \Gamma; \tau \vdash_{\text{styp}} \mathsf{if}\ 0\ s_1\ s_2$ provides $\cdot; \Upsilon; \Gamma; \tau \vdash_{\text{styp}} s_2$.
- DS3.5: Inverting $\cdot; \Upsilon; \Gamma; \tau \vdash_{\text{styp}} \mathsf{if}\ i\ s_1\ s_2$ provides $\cdot; \Upsilon; \Gamma; \tau \vdash_{\text{styp}} s_1$.
- DS3.6: Inverting $\cdot; \Upsilon; \Gamma; \tau \vdash_{\text{styp}} \mathsf{while}\ e\ s$ provides $\cdot; \Upsilon; \Gamma; \tau \vdash_{\text{styp}} s$ and $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} e : \text{int}$. Typing Well-Formedness Lemma provides $\vdash_{\text{wf}} \cdot; \Upsilon; \Gamma$, so $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} 0 : \text{int}$. With these results, we can use SS3.5, SS3.3, and SS3.1 to derive $\cdot; \Upsilon; \Gamma; \tau \vdash_{\text{styp}} \mathsf{if}\ e\ (s; \mathsf{while}\ e\ s)\ 0$.
- DS3.7: Inverting $\cdot; \Upsilon; \Gamma; \tau'' \vdash_{\text{styp}} \mathsf{open}\ (\mathsf{pack}\ \tau', v\ \mathsf{as}\ \exists^{\phi}\alpha{:}\kappa.\tau)\ \mathsf{as}\ \alpha, x;\ s$ provides $\alpha{:}\kappa; \Upsilon; \Gamma, x{:}\tau; \tau'' \vdash_{\text{styp}} s$, $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v : \tau[\tau'/\alpha]$, $\cdot \vdash_{\text{ak}} \tau' : \kappa$, and $\cdot \vdash_{\text{k}} \tau'' : \mathtt{A}$. So Substitution Lemma 8 provides $\cdot; \Upsilon; (\Gamma[\tau'/\alpha]), x{:}\tau[\tau'/\alpha]; \tau''[\tau'/\alpha] \vdash_{\text{styp}} s[\tau'/\alpha]$. Applying Useless Substitution Lemmas 1 and 2 (using Typing Well-Formedness Lemma for $\vdash_{\text{wf}} \cdot; \Upsilon; \Gamma$) provides $\cdot; \Upsilon; \Gamma, x{:}\tau[\tau'/\alpha]; \tau'' \vdash_{\text{styp}} s[\tau'/\alpha]$. So SS3.6 lets us derive $\cdot; \Upsilon; \Gamma; \tau'' \vdash_{\text{styp}} \mathsf{let}\ x = v;\ s[\tau'/\alpha]$, as desired.
- DS3.8: Inverting $\cdot; \Upsilon; \Gamma; \tau'' \vdash_{\text{styp}} \mathsf{open}\ xp\ \mathsf{as}\ \alpha, *x';\ s$ provides $\alpha{:}\kappa; \Upsilon; \Gamma, x'{:}\tau*; \tau'' \vdash_{\text{styp}} s$, $\Upsilon; x\cdot \vdash \text{gettype}(\Gamma(x), p, \exists^{\&}\alpha{:}\kappa.\tau)$, and $\cdot \vdash_{\text{k}} \tau'' : \mathtt{A}$. Heap-Object Safety Lemmas 1 and 3 provide $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} \mathsf{pack}\ \tau', v\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau : \exists^{\&}\alpha{:}\kappa.\tau$. Inverting this result provides $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} v : \tau[\tau'/\alpha]$ and $\cdot \vdash_{\text{ak}} \tau' : \kappa$. So Substitution Lemma 8 provides $\cdot; \Upsilon; (\Gamma[\tau'/\alpha]), x'{:}\tau*[\tau'/\alpha]; \tau''[\tau'/\alpha] \vdash_{\text{styp}} s[\tau'/\alpha]$. Applying Useless Substitution Lemmas 1 and 2 (using Typing Well-Formedness Lemma for $\vdash_{\text{wf}} \cdot; \Upsilon; \Gamma$) provides $\cdot; \Upsilon; \Gamma, x{:}\tau*[\tau'/\alpha]; \tau'' \vdash_{\text{styp}} s[\tau'/\alpha]$.

  Let $\Upsilon' = \Upsilon, xp{:}\tau'$, $\Gamma' = \Gamma$, and $H' = H$. It is impossible that Υ' is not an extension of Υ, because that would violate the assumption $H \vdash_{\text{refp}} \Upsilon$. It may be that $xp \in \text{Dom}(\Upsilon)$ in which case $\Upsilon' = \Upsilon$, which is fine. Applying Term Weakening Lemma 1 to $\Upsilon; x\cdot \vdash \text{gettype}(\Gamma(x), p, \exists^{\&}\alpha{:}\kappa.\tau)$ provides $\Upsilon'; x\cdot \vdash \text{gettype}(\Gamma(x), p, \exists^{\&}\alpha{:}\kappa.\tau)$. Applying Path Extension Lemma 2 to this result provides $\Upsilon'; x\cdot \vdash \text{gettype}(\Gamma(x), p\mathtt{u}, \tau[\tau'/\alpha])$. So SL3.1 and SR3.6 let us derive $\cdot; \Upsilon'; \Gamma' \vdash_{\text{rtyp}} \&xp\mathtt{u} : \tau[\tau'/\alpha]$. Applying Term Weakening Lemma to $\cdot; \Upsilon; \Gamma, x{:}\tau*[\tau'/\alpha]; \tau'' \vdash_{\text{styp}} s[\tau'/\alpha]$ provides $\cdot; \Upsilon'; \Gamma', x{:}\tau*[\tau'/\alpha]; \tau'' \vdash_{\text{styp}} s[\tau'/\alpha]$. So SS3.6 lets us derive $\cdot; \Upsilon'; \Gamma'; \tau'' \vdash_{\text{styp}} \mathsf{let}\ x' = \&xp\mathtt{u};\ s[\tau'/\alpha]$, as desired.
- DS3.9–11: These cases use inductive arguments similar to cases DR3.8–10. Again, the Term Weakening Lemma allows unchanged contained terms to type-check under Γ' and Υ'. For binding forms (let and open), α-conversion (of x) ensures that $\Gamma', x{:}\tau'$ makes sense.

Lemma A.14 (Term Progress). *Suppose* $\Upsilon; \Gamma \vdash_{\text{htyp}} H : \Gamma$ *and* $H \vdash_{\text{refp}} \Upsilon$.

- *If* $\cdot; \Upsilon; \Gamma \vdash_{\text{ltyp}} e : \tau$, *then* e *has the form* xp *or there exists an* H' *and* e' *such that* $H; e \xrightarrow{\text{l}} H'; e'$.
- *If* $\cdot; \Upsilon; \Gamma \vdash_{\text{rtyp}} e : \tau$, *then* e *is some value* v *or there exists an* H' *and* e' *such that* $H; e \xrightarrow{\text{r}} H'; e'$.
- *If* $\cdot; \Upsilon; \Gamma; \tau \vdash_{\text{styp}} s$, *then* s *has the form* v *or* return v, *or there exists an* H' *and* s' *such that* $H; s \xrightarrow{\text{s}} H'; s'$.

Proof: The proof is by simultaneous induction on the assumed typing derivations, proceeding by cases on the last rule used:

- SL3.1: e has the form xp.
- SL3.2: By induction, if e' (where $e = *e'$) is not a value, it can take a step, so DL3.3 applies. Else e' is a value with a pointer type, so the Canonical Forms Lemma provides it has the form $\&xp$. So DL3.2 applies.
- SL3.3: By induction, if e' (where $e = e'.0$) is not some xp, it can take a step, so DL3.4 applies. Else e' is some xp, so DL3.1 applies.
- SL3.4: This case is analogous to the previous one.
- SR3.1: Heap Safety Lemma 3 provides $\text{get}(H(x), p, v)$ for some v, so DR3.1 applies.
- SR3.2: This case is analogous to SL3.2, using DR3.10 and DR3.3 in place of DL3.3 and DL3.2.
- SR3.3: By induction, if e' (where $e = e'.0$) is not a value, it can take a step, so DR3.10 applies. Else e' is a value with a product type, so the Canonical Forms Lemma provides it has the form (v_0, v_1). So DR3.4 applies.
- SR3.4: This case is analogous to the previous one.
- SR3.5: e is a value.
- SR3.6: By induction, if e' (where $e = \&e'$) is not some xp, it can take a step, so DR3.9 applies. Else e is a value.
- SR3.7: Let $e = (e_0, e_1)$. If e_0 is not a value, or e_0 is a value but e_1 is not a value, then induction ensures the nonvalue can take a step, so DR3.10 applies. Else e is a value.

- SR3.8: Let $e = (e_1{=}e_2)$. If e_1 is not some xp, then induction ensures e_1 can take a step, so DR3.9 applies. Else if e_2 is not a value, then induction ensures e_2 can take a step, so DR3.10 applies. Else the typing derivation and Heap-Object Safety Lemma 3 provide the hypothesis to DR3.2.

- SR3.9: Let $e = e_1(e_2)$. By induction, if e_1 is not a value or e_1 is a value and e_2 is not a value, then the nonvalue can take a step and DR3.10 applies. Else, e_1 is a value with a function type, so the Canonical Forms Lemma provides it is a function. So DR3.5 applies.

- SR3.10: By induction, if s is not v or $\mathsf{return}\ v$, then it can take a step so DR3.8 applies. Else s is v or $\mathsf{return}\ v$. Inspection of $\vdash_{\mathrm{ret}} s$ (provided by inversion of the typing derivation) shows the former case is impossible). In the latter case, DR3.6 applies.

- SR3.11: Let $e = e'[\tau]$. By induction, if e' is not a value, it can take step, so DR3.10 applies. Else it is a value with a universal type, so the Canonical Forms Lemma ensures it is a polymorphic value. So DR.27 applies.

- SR3.12: By induction, if the expression inside the package is not a value, it can take a step, so DR3.10 applies. Else e is a value.

- SR3.13: e is a value.

- SR3.14: e is a value.

- SS3.1: By induction, if e is not a value, it can take a step so DS3.9 applies. Else s is a value.

- SS3.2: By induction, if e is not a value, it can take a step so DS3.9 applies. Else s has the form $\mathsf{return}\ v$.

- SS3.3: By induction, s can take a step, is some v, or has the form $\mathsf{return}\ v$. In the first case, DS3.10 applies. In the second case, DS3.2 applies. In the third case, DS3.3 applies.

- SS3.4: DS3.6 applies.

- SS3.5: By induction, if e is not a value, it can take a step, so DS3.9 applies. Else e is a value of type int, so the Canonical Forms Lemma ensures it is some i. So either DS3.4 or DS3.5 applies.

- SS3.6: By induction, if e is not a value, it can take a step, so DS3.9 applies. Else DS3.1 applies.

- SS3.7: By induction, if e is not a value, it can take a step, so DS3.9 applies. Else e is a value with an existential type, so the Canonical Forms Lemma ensures it is an existential package. So DS3.7 applies.

- SS3.8: By induction, if e is not of the form xp, it can take a step, so DS3.11 applies. Else e has the form xp and $\Upsilon; x\cdot \vdash \mathrm{gettype}(\Gamma(x), p, \exists^{\&}\alpha{:}\kappa.\tau')$. So Heap-Object Safety Lemma 3 provides there exists some v such that $\mathrm{get}(H(x), p, v)$ and $\cdot; \Upsilon; \Gamma \vdash_{\mathrm{rtyp}} v : \exists^{\&}\alpha{:}\kappa.\tau'$. So the Canonical Forms Lemma provides v has the form $\mathsf{pack}\ \tau'', v'\ \mathsf{as}\ \exists^{\&}\alpha{:}\kappa.\tau'$. So DS3.8 applies.

It is straightforward to check that the preservation and progress properties stated in the proof of the Type Safety Theorem are corollaries to the Return Preservation Lemma, the Term Preservation Lemma, and the Term Progress Lemma. These lemmas apply given the hypotheses of $\vdash_{\mathrm{prog}} P$ and the conclusions of the preservation lemmas suffice to conclude $\vdash_{\mathrm{prog}} P'$. The lemmas are stronger (e.g., the static context is an extension) because of their inductive proofs.

Appendix B

Chapter 4 Safety Proof

This appendix proves Theorem 4.2, which we repeat here:

Definition 4.1. *State $S_G; S; s$ is stuck if s is not of the form* return v *and there are no S'_G, S', and s' such that $S_G; S; s \xrightarrow{s} S'_G; S'; s'$.*

Theorem 4.2 (Type Safety). *If $\cdot;\cdot;\cdot;\cdot;\emptyset;\tau \vdash_{\text{styp}} s$, $\vdash_{\text{ret}} s$, s contains no pop statements, and $\cdot;\cdot;s \xrightarrow{s}{}^{*} S'_G; S'; s'$ (where $\xrightarrow{s}{}^{*}$ is the reflexive, transitive closure of $\xrightarrow{s}$), then $S'_G; S'; s'$ is not stuck.*

Before presenting and proving the necessary lemmas in "bottom-up" order, we summarize the structure of the argument and explain how the lemmas imply the Type Safety Theorem. Because the theorem's assumptions imply $\vdash_{\text{prog}} \cdot;\cdot;s$, a simple induction on the number of steps taken shows that it suffices to establish preservation (if $\vdash_{\text{prog}} S_G; S; s$ and $S_G; S; s \xrightarrow{s} S'_G; S'; s'$, then $\vdash_{\text{prog}} S'_G; S'; s'$) and progress (if $\vdash_{\text{prog}} S_G; S; s$, then s is not stuck). To prove these properties inductively, we need analogous lemmas for right-expressions and left-expressions as in Chapter 3, but memory allocation complicates matters.

Chapter 4 explains why the hypotheses for the $\vdash_{\text{prog}}$ rule type-check s under the capability $\emptyset$ and require $R_P \vdash_{\text{spop}} s$. But we must change these restrictions to apply the induction hypothesis when s has the form $s_1;$ pop i. After all, we should allow access to i within s_1 and s_1 should not deallocate i. The necessary generalization of $\vdash_{\text{prog}}$, defined in Figure B.1, conceptually partitions the live regions S such that $S = S_E S_P$. A statement or expression is acceptable if it type-checks under a capability consisting of the regions in S_E and deallocates the regions in S_P (subject to the other restrictions of $\vdash_{\text{spop}}$ and $\vdash_{\text{epop}}$). Otherwise, the judgments used in the statement of the Type and Pop Preservation Lemma and the Type and Pop Progress Lemma are like $\vdash_{\text{prog}}$.

When $S_G; S_E S_P; s$ becomes $S'_G; S'_E S'_P; s'$, a region might be deallocated (shrinking S_P and growing S_G), a region might be allocated (growing S_P), a live location

might be mutated (changing S_E or S_P but not its type), or the heap might remain unchanged. The Type and Pop Preservation Lemma demonstrates that the type context that the heap induces is strong enough for the resulting state to be well-typed in all of these situations. The most interesting use of the induction hypothesis is when s is $s_1; \mathsf{pop}\ i$ because the proof conceptually "shifts" the deepest region in S_P to the shallowest region in S_E to apply induction to s_1.

Applying the Type and Pop Preservation Lemma with $S_P = \cdot$ in conjunction with the Return Preservation Lemma implies the statement of preservation we need. Similarly, applying the Type and Pop Progress Lemma with $S_P = \cdot$ in conjunction with $\vdash_{\text{ret}} s$ implies the statement of progress we need.

The Type and Pop Progress Lemma relies on the Canonical Forms Lemma (as usual), the Heap-Object Safety Lemma (for progress results involving paths; these are much simpler than in Chapter 3), and the Access Control Lemma. This last lemma ensures the $\vdash_{\text{acc}}$ hypotheses suffice to prevent programs from trying to access S_G (and therefore becoming stuck). In languages where programs may access all locations in scope, such lemmas are unnecessary.

Proving the Type and Pop Preservation Lemma requires several auxiliary lemmas. The New-Region Preservation Lemma dismisses a technical point for cases that allocate regions: Statements like s_1 in $\mathsf{region}\ \rho, x\ s_1$ assume the constraint $(i_1 \cup \ldots \cup i_n) < \rho$, but the typing context after the step provides $i_1 < i_2, \ldots, i_n < i$ (where we substitute $\mathrm{S}(i)$ for ρ). This lemma proves these constraints suffice to type-check s_1 (after the substitution). The Subtyping Preservation Lemma proves results that would hold by inversion were it not for rules SR4.17 and SL4.5. Like for progress, the Heap-Object Safety Lemmas provide results about paths and the associated relations (get, set, and gettype). The proofs are simple without the reference patterns of Chapter 3. The Values Effectless Lemma ensures $C \vdash_{\text{rtyp}} v : \tau$ and $R \vdash_{\text{epop}} v$ means v type-checks regardless of the current capability (intuitively, "evaluating" values does not access the heap) and v contains no pop statements. This lemma is crucial for allowing values to escape scope.

The other auxiliary lemmas for preservation are more conventional. The Term Substitution Lemma provides the preservation result we need for dynamic steps that use type substitution. The Heap-Type Well-Formedness Lemma just describes what we can assume about heap locations by inverting the $\vdash_{\text{htyp}}$ judgments. The Typing Well-Formedness Lemma lets us use typing judgments to conclude context well-formedness and some types' kinds. The Type Substitution Lemma provides type-level results we need to prove the Typing Well-Formedness Lemma and the Term Substitution Lemma. The eighth such lemma needs the Type Canonical Forms Lemma, which is rather obvious. The remaining lemmas (Commuting Substitutions, Useless Substitution, and various weakening lemmas) serve the same purpose they did in Chapter 3. For each, we need to prove analogous results for

effects, constraints, and types. The Commuting Substitutions Lemma for effects is interesting because we must prove regions($\tau_1[\tau_2/\alpha]$) = (regions(τ_1))$[\tau_2/\alpha]$. The weakening lemmas always use the semantic notion of stronger effects and constraints (the $\vdash_{\text{eff}}$ judgments) rather than a less useful notion of syntactic extension.

Lemma B.1 (Context Weakening).

1. *If* $R; \Delta \vdash_{\text{wf}} \epsilon$ *and* $R \subseteq R'$, *then* $R'; \Delta\Delta' \vdash_{\text{wf}} \epsilon$.
2. *If* $R; \Delta \vdash_{\text{wf}} \gamma$ *and* $R \subseteq R'$, *then* $R'; \Delta\Delta' \vdash_{\text{wf}} \gamma$.
3. *If* $R; \Delta \vdash_{\text{k}} \tau : \kappa$ *and* $R \subseteq R'$, *then* $R'; \Delta\Delta' \vdash_{\text{k}} \tau : \kappa$.
4. *If* $R; \Delta \vdash_{\text{wf}} \Gamma$ *and* $R \subseteq R'$, *then* $R'; \Delta\Delta' \vdash_{\text{wf}} \Gamma$.
5. *If* $\gamma \vdash_{\text{eff}} \epsilon_1 \Leftarrow \epsilon_2$ *and* $\gamma' \vdash_{\text{eff}} \gamma$, *then* $\gamma' \vdash_{\text{eff}} \epsilon_1 \Leftarrow \epsilon_2$.
6. *If* $\gamma; \epsilon \vdash_{\text{acc}} r$, $\gamma' \vdash_{\text{eff}} \gamma$, *and* $\gamma' \vdash_{\text{eff}} \epsilon \Leftarrow \epsilon'$, *then* $\gamma'; \epsilon' \vdash_{\text{acc}} r$.
7. *If* $\gamma_2 \vdash_{\text{eff}} \gamma_1$ *and* $\gamma_3 \vdash_{\text{eff}} \gamma_2$, *then* $\gamma_3 \vdash_{\text{eff}} \gamma_1$.

Proof:

1. By induction on the derivation of $R; \Delta \vdash_{\text{wf}} \epsilon$
2. By induction on the derivation of $R; \Delta \vdash_{\text{wf}} \gamma$, using the previous lemma
3. By induction on the derivation of $R; \Delta \vdash_{\text{k}} \tau : \kappa$, using the previous lemmas
4. By induction on the derivation of $R; \Delta \vdash_{\text{wf}} \Gamma$, using the previous lemma
5. By induction on the derivation of $\gamma \vdash_{\text{eff}} \epsilon_1 \Leftarrow \epsilon_2$: The interesting case is when $\gamma = \gamma_1, \epsilon_1 < \epsilon_2, \gamma_2$ and the derivation ends with the corresponding axiom. In this case, $\gamma' \vdash_{\text{eff}} \gamma$ ensures $\gamma' \vdash_{\text{eff}} \epsilon_1 \Leftarrow \epsilon_2$.
6. By inspection of the $\gamma; \epsilon \vdash_{\text{acc}} r$ judgment, using the previous lemma and the $\vdash_{\text{eff}}$ rule for transitivity
7. By induction on the derivation of $\gamma_2 \vdash_{\text{eff}} \gamma_1$, using Context Weakening Lemma 5

Lemma B.2 (Term Weakening). *Suppose* $\vdash_{\text{wf}} R'; \Delta\Delta'; \Gamma\Gamma'; \gamma'; \epsilon'$, $R \subseteq R'$, $\gamma' \vdash_{\text{eff}} \gamma$, *and* $\gamma' \vdash_{\text{eff}} \epsilon \Leftarrow \epsilon'$.

1. *If* $R; \Delta; \Gamma; \gamma; \epsilon \vdash_{\text{ltyp}} e : \tau, r$, *then* $R'; \Delta\Delta'; \Gamma\Gamma'; \gamma'; \epsilon' \vdash_{\text{ltyp}} e : \tau, r$.

2. *If* $R; \Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} e : \tau$, *then* $R'; \Delta\Delta'; \Gamma\Gamma'; \gamma'; \epsilon' \vdash_{\text{rtyp}} e : \tau$.

3. *If* $R; \Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{\text{styp}} s$, *then* $R'; \Delta\Delta'; \Gamma\Gamma'; \gamma'; \epsilon'; \tau \vdash_{\text{styp}} s$.

Proof: By simultaneous induction on the assumed typing derivations, proceeding by cases on the last rule in the derivation:

- SS4.1–5: These cases follow from induction.
- SS4.6–8: These cases follow from induction and Context Weakening Lemma 3. The induction hypothesis applies because of α-conversion, implicit reordering of Δ and Γ, the fact that $\gamma' \vdash_{\text{eff}} \gamma$ implies $\gamma', \epsilon{<}\rho \vdash_{\text{eff}} \gamma, \epsilon{<}\rho$, and the fact that $\gamma' \vdash_{\text{eff}} \epsilon \Leftarrow \epsilon'$ implies $\gamma' \vdash_{\text{eff}} \epsilon \cup \rho \Leftarrow \epsilon' \cup \rho$.
- SS4.9: This case follows from induction, which applies because $\gamma' \vdash_{\text{eff}} \epsilon \Leftarrow \epsilon'$ implies $\gamma' \vdash_{\text{eff}} \epsilon \cup i \Leftarrow \epsilon' \cup i$.
- SL4.1: This case is trivial because $\Gamma\Gamma'(x) = \Gamma(x)$.
- SL4.2–4: These cases follow from induction.
- SL4.5: This case follows from induction and Context Weakening Lemmas 3 and 6.
- SR4.1: This case is trivial because $\Gamma\Gamma'(x) = \Gamma(x)$.
- SR4.2–7: These cases follow from induction. (SR4.5 is trivial.)
- SR4.8: This case follows from induction and Context Weakening Lemma 6.
- SR4.9: This case follows from induction and Context Weakening Lemma 5.
- SR4.10: This case follows from induction.
- SR4.11–12: These cases follow from induction and Context Weakening Lemmas 3 and 7.
- SR4.13–14: These cases are like cases SS4.6–8. For SR4.13, the induction hypothesis applies to the function body using the function's explicit effect for ϵ and e'. This explicit effect has no connection to the ϵ or ϵ' used (but ignored) to type-check the function.
- SR4.15: This case follows from induction and Context Weakening Lemma 6.
- SR4.16: This case is trivial because $i \in R'$.

- SR4.17: This case is like SL4.5.

Lemma B.3 (Heap-Type Weakening). *Suppose* $R \subseteq R'$, $R'; \cdot \vdash_{\text{wf}} \Gamma\Gamma'$, $R'; \cdot \vdash_{\text{wf}} \gamma'$, *and* $\gamma' \vdash_{\text{eff}} \gamma$.

1. *If* $R; \Gamma; \gamma; i \vdash_{\text{htyp}} H : \Gamma''$, *then* $R'; \Gamma\Gamma'; \gamma'; i \vdash_{\text{htyp}} H : \Gamma''$.

2. *If* $R; \Gamma; \gamma \vdash_{\text{htyp}} S : \Gamma''$, *then* $R'; \Gamma\Gamma'; \gamma' \vdash_{\text{htyp}} S : \Gamma''$.

Proof: The first proof is by induction on the derivation of $R; \Gamma; \gamma; i \vdash_{\text{htyp}} H : \Gamma''$ using Term Weakening Lemma 2. The second proof is by induction on the derivation of $R; \Gamma; \gamma \vdash_{\text{htyp}} S : \Gamma''$, using the first lemma.

Lemma B.4 (Useless Substitution). *Suppose* $\alpha \notin \text{Dom}(\Delta)$.

1. *If* $R; \Delta \vdash_{\text{wf}} \epsilon$, *then* $\epsilon[\tau/\alpha] = \epsilon$.

2. *If* $R; \Delta \vdash_{\text{wf}} \gamma$, *then* $\gamma[\tau/\alpha] = \gamma$.

3. *If* $R; \Delta \vdash_{\text{k}} \tau' : \kappa$, *then* $\tau'[\tau/\alpha] = \tau'$.

4. *If* $R; \Delta \vdash_{\text{wf}} \Gamma$, *then* $\Gamma[\tau/\alpha] = \Gamma$.

Proof: Each proof is by induction on the assumed derivation, appealing to the definition of substitution and the preceding lemmas as necessary.

Lemma B.5 (Type Canonical Forms). *If* $R; \Delta \vdash_{\text{k}} \tau : \mathtt{R}$, *then* $\tau = S(i)$ *for some* $i \in R$ *or* $\tau = \alpha$ *for some* $\alpha \in \text{Dom}(\Delta)$.

Proof: By inspection of the $\vdash_{\text{k}}$ rules

Lemma B.6 (Commuting Substitutions). *Suppose* $\beta \neq \alpha$ *and* β *is not free in* τ_2.

1. $\epsilon[\tau_1/\beta][\tau_2/\alpha] = \epsilon[\tau_2/\alpha][\tau_1[\tau_2/\alpha]/\beta]$

2. $\gamma[\tau_1/\beta][\tau_2/\alpha] = \gamma[\tau_2/\alpha][\tau_1[\tau_2/\alpha]/\beta]$

3. $\tau_0[\tau_1/\beta][\tau_2/\alpha] = \tau_0[\tau_2/\alpha][\tau_1[\tau_2/\alpha]/\beta]$

Proof:

1. By induction on the structure of ϵ (assuming set equalities as usual): The cases where ϵ is $\emptyset$, i, or some α' that is neither α nor β are trivial. The case where $\epsilon = \epsilon_1 \cup \epsilon_2$ is by induction. If $\epsilon = \alpha$, then both substitutions produce regions(τ_2); for the right side, we rely on the assumption that β is not free in τ_2 and the definition of regions for the outer substitution to be useless. If $\epsilon = \beta$, the left substitution produces regions(τ_1)[τ_2/α] and the right produces regions($\tau_1[\tau_2/\alpha]$). An inductive argument on the structure of τ_1 ensures these sets are the same.

2. By induction on the structure of γ, using the previous lemma

3. By induction on the structure of τ_0: The cases for int and S(i) are trivial. The cases for pair types, pointer types, and handle types are by induction. The case for function types is by induction and Commuting Substitutions Lemma 1. The cases for quantified types are by induction and Commuting Substitutions Lemma 2. The case for α' that is neither α nor β is trivial. If $\tau_0 = \alpha$, then both substitutions produce τ_2; for the right side, we rely on the assumption that β is not free in τ_2 for the outer substitution to be useless. If $\tau_0 = \beta$, both substitutions produce $\tau_1[\tau_2/\alpha]$.

Lemma B.7 (Type Substitution). *Suppose* $R; \Delta \vdash_{\mathrm{k}} \tau : \kappa$.

1. $R; \Delta \vdash_{\mathrm{wf}} \mathrm{regions}(\tau)$
2. *If* $R; \Delta, \alpha{:}\kappa \vdash_{\mathrm{wf}} \epsilon$, *then* $R; \Delta \vdash_{\mathrm{wf}} \epsilon[\tau/\alpha]$.
3. *If* $R; \Delta, \alpha{:}\kappa \vdash_{\mathrm{wf}} \gamma$, *then* $R; \Delta \vdash_{\mathrm{wf}} \gamma[\tau/\alpha]$.
4. *If* $R; \Delta, \alpha{:}\kappa \vdash_{\mathrm{k}} \tau' : \kappa'$, *then* $R; \Delta \vdash_{\mathrm{k}} \tau'[\tau/\alpha] : \kappa'$.
5. *If* $R; \Delta, \alpha{:}\kappa \vdash_{\mathrm{wf}} \Gamma$, *then* $R; \Delta \vdash_{\mathrm{wf}} \Gamma[\tau/\alpha]$.
6. *If* $\vdash_{\mathrm{wf}} R; \Delta, \alpha{:}\kappa; \Gamma; \gamma; \epsilon$, *then* $\vdash_{\mathrm{wf}} R; \Delta; \Gamma[\tau/\alpha]; \gamma[\tau/\alpha]; \epsilon[\tau/\alpha]$.
7. *If* $\gamma \vdash_{\mathrm{eff}} \epsilon_1 \Leftarrow \epsilon_2$, *then* $\gamma[\tau/\alpha] \vdash_{\mathrm{eff}} \epsilon_1[\tau/\alpha] \Leftarrow \epsilon_2[\tau/\alpha]$.
8. *If* $\gamma; \epsilon \vdash_{\mathrm{acc}} r$ *and* $R; \Delta, \alpha{:}\kappa \vdash_{\mathrm{k}} r : \mathtt{R}$, *then* $\gamma[\tau/\alpha]; \epsilon[\tau/\alpha] \vdash_{\mathrm{acc}} r[\tau/\alpha]$.
9. *If* $\gamma \vdash_{\mathrm{eff}} \gamma'$, *then* $\gamma[\tau/\alpha] \vdash_{\mathrm{eff}} \gamma'[\tau/\alpha]$.

Proof:

1. By induction on the assumed kinding derivation: Cases in which regions(τ) is $\emptyset$ are trivial. The cases for subkinding, pair types, pointer types, and handle types are by induction. The case for function types follows immediately from the rule's right-most hypothesis. The cases for type variables and singleton types follow from the rules' assumptions and the definition of $\vdash_{\text{wf}}$. For $\forall\alpha{:}\kappa'[\gamma].\tau'$ or $\exists\alpha{:}\kappa'[\gamma].\tau'$, induction provides $R; \Delta, \alpha{:}\kappa' \vdash_{\text{wf}} \text{regions}(\tau')$. A trivial induction on $\vdash_{\text{wf}}$ shows that if $R; \Delta, \alpha{:}\kappa' \vdash_{\text{wf}} \epsilon$ and $\alpha \notin \epsilon$, then $R; \Delta \vdash_{\text{wf}} \epsilon$. Hence $R; \Delta \vdash_{\text{wf}} \text{regions}(\tau') - \alpha$, as desired.

2. By induction on the derivation of $R; \Delta, \alpha{:}\kappa \vdash_{\text{wf}} \gamma$: The previous lemma ensures the interesting case, when $\epsilon = \alpha$.

3. By induction on the derivation of $R; \Delta, \alpha{:}\kappa \vdash_{\text{wf}} \gamma$, using the previous lemma

4. By induction on the derivation of $R; \Delta, \alpha{:}\kappa \vdash_{\text{k}} \tau' : \kappa'$: Most cases are immediate or by induction. The case for function types also uses Type Substitution Lemma 2. The case for quantified types also uses Type Substitution Lemma 3 and implicit reordering of type-variable contexts.

5. By induction on the derivation of $R; \Delta, \alpha{:}\kappa \vdash_{\text{wf}} \Gamma$, using the previous lemma

6. This lemma is a corollary to Type Substitution Lemmas 2, 3, and 5.

7. By induction on the derivation of $\gamma \vdash_{\text{eff}} \epsilon_1 \Leftarrow \epsilon_2$: The two axioms follow from the definition of substitution. The other cases are by induction.

8. By inspection of the derivation of $\gamma; \epsilon \vdash_{\text{acc}} r$, using the previous lemma: Type Substitution Lemma 3 and the Type Canonical Forms Lemma ensure the form of $r[\tau/\alpha]$ is appropriate for $\vdash_{\text{acc}}$.

9. By induction on the derivation of $\gamma \vdash_{\text{eff}} \gamma'$, using Type Substitution Lemma 7

Lemma B.8 (Typing Well-Formedness).

1. *If* $\vdash \text{gettype}(\tau', p, \tau)$ *and* $C_R; C_\Delta \vdash_{\text{k}} \tau' : \mathtt{A}$, *then* $C_R; C_\Delta \vdash_{\text{k}} \tau : \mathtt{A}$.

2. *If* $C \vdash_{\text{ltyp}} e : \tau, r$, *then* $\vdash_{\text{wf}} C$, $C_R; C_\Delta \vdash_{\text{k}} \tau : \mathtt{A}$, *and* $C_R; C_\Delta \vdash_{\text{k}} r : \mathtt{R}$.

3. *If* $C \vdash_{\text{rtyp}} e : \tau$, *then* $\vdash_{\text{wf}} C$ *and* $C_R; C_\Delta \vdash_{\text{k}} \tau : \mathtt{A}$.

4. *If* $C; \tau \vdash_{\text{styp}} s$, *then* $\vdash_{\text{wf}} C$. *If* $C; \tau \vdash_{\text{styp}} s$ *and* $\vdash_{\text{ret}} s$, *then* $C_R; C_\Delta \vdash_{\text{k}} \tau : \mathtt{A}$.

Proof: The first proof is by induction on the $\vdash \mathrm{gettype}(\tau, p, \tau')$ derivation. If $p = \cdot$, the result is immediate, else inversion of the kinding derivation ensures the induction hypothesis applies. The remaining proofs are by simultaneous induction on the assumed typing derivations, proceeding by cases on the last rule used. Several of the cases that invert kinding derivations implicitly cover two cases because the last step may subsume kind `B` to `A`.

- SL4.1, SR4.1: These cases follow from the definition of $C_R; C_\Delta \vdash_{\mathrm{wf}} \Gamma$ (for the kind of τ' and r) and Typing Well-Formedness Lemma 1.
- SL4.2–4, SR4.2–4: This case follows from induction and inversion of the kinding derivation for the type of the term used in the hypothesis.
- SL4.5, SR4.17: These cases follow from induction on the left hypothesis, inversion of the kinding derivation for its type, the right hypothesis, and (for SR4.17) the kinding rule for pointer types.
- SR4.5: This case is trivial.
- SR4.6–7: These cases follow from induction and the kinding rules for pointer and pair types.
- SR4.8: This case follows from induction, using the middle hypothesis.
- SR4.9: This case follows from induction, using the left hypothesis and inversion of the kinding derivation for function types.
- SR4.10: This case follow from induction.
- SR4.11: This case follows from induction on the left hypothesis, inversion of the kinding derivation for the quantified type, and Type Substitution Lemma 4 (using the middle hypothesis from the typing derivation).
- SR4.12: This case follows from the right hypothesis and induction on the left hypothesis.
- SR4.13–14: These cases are trivial.
- SR4.15: This case follows from induction, inversion of the kinding derivation for the handle type, and the kinding rule for pointer types.
- SR4.16: This case follows from the kinding rule for singleton types.
- SS4.1–5: These cases follow from induction. Note that the $\vdash_{\mathrm{ret}}$ obligation holds vacuously for SS4.1 and SS4.4.

- SS4.6–7: These cases follow from induction on the expression-typing hypothesis and the kinding hypothesis for τ.
- SS4.8: This case follows immediately from the hypotheses.
- SS4.9: This case follows from induction and the fact that if $\epsilon \cup i$ is well-formed, then ϵ is well-formed.

Lemma B.9 (Heap-Type Well-Formedness). *If* $R; \Gamma'; \gamma \vdash_{\text{htyp}} S : \Gamma$, *then* $x \in \text{Dom}(\Gamma)$ *if and only if* x *is in some* H *in* S. *If* $\Gamma(x) = (\tau, r)$ *then* $r = \text{S}(i))$ *and* S *has the form* $S_1, i : H_1, x \mapsto v, H_2, S_2$ *where* $R; \cdot; \Gamma'; \gamma; \emptyset \vdash_{\text{rtyp}} v : \tau$ *and* $\cdot \vdash_{\text{epop}} v$.

Proof: By induction on the $\vdash_{\text{htyp}}$ derivations

Lemma B.10 (Term Substitution).

1. *If* $\vdash_{\text{ret}} s$, *then* $\vdash_{\text{ret}} s[\tau/\alpha]$.
2. *If* $R \vdash_{\text{spop}} s$, *then* $R \vdash_{\text{spop}} s[\tau/\alpha]$.
   If $R \vdash_{\text{epop}} e$, *then* $R \vdash_{\text{epop}} e[\tau/\alpha]$.
3. *If* $\vdash \text{gettype}(\tau_1, p, \tau_2)$, *then* $\vdash \text{gettype}(\tau_1[\tau/\alpha], p, \tau_2[\tau/\alpha])$.
4. *Suppose* $R; \Delta \vdash_{\text{k}} \tau : \kappa$.
   If $R; \Delta, \alpha{:}\kappa; \Gamma; \gamma; \epsilon \vdash_{\text{ltyp}} e : \tau', r$,
   then $R; \Delta; \Gamma[\tau/\alpha]; \gamma[\tau/\alpha]; \epsilon[\tau/\alpha] \vdash_{\text{ltyp}} e[\tau/\alpha] : \tau'[\tau/\alpha], r[\tau/\alpha]$.
   If $R; \Delta, \alpha{:}\kappa; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} e : \tau'$,
   then $R; \Delta; \Gamma[\tau/\alpha]; \gamma[\tau/\alpha]; \epsilon[\tau/\alpha] \vdash_{\text{rtyp}} e[\tau/\alpha] : \tau'[\tau/\alpha]$.
   If $R; \Delta, \alpha{:}\kappa; \Gamma; \gamma; \epsilon; \tau' \vdash_{\text{styp}} s$,
   then $R; \Delta; \Gamma[\tau/\alpha]; \gamma[\tau/\alpha]; \epsilon[\tau/\alpha]; \tau'[\tau/\alpha] \vdash_{\text{styp}} s[\tau/\alpha]$.

Proof:

1. By induction on the derivation of $\vdash_{\text{ret}} s$
2. By simultaneous induction on the derivations of $R \vdash_{\text{spop}} s$ and $R \vdash_{\text{epop}} e$
3. By induction on the derivation of $\vdash \text{gettype}(\tau_1, p, \tau_2)$
4. By simultaneous induction on the assumed derivations, proceeding by cases on the last rule used: In each case, we satisfy the hypotheses of the rule after substitution and then use the rule to derive the desired result. So for each case, we just list the lemmas and arguments needed to conclude the necessary hypotheses. Cases SR4.11 and SR4.12 use the Commuting Substitutions Lemma just like cases SR3.11 ans SR3.12 in Chapter 3; see there for details.

- SL4.1: the definition of substitution, Term Substitution Lemma 3, and Type Substitution Lemma 6
- SL4.2–4: induction
- SL4.5: induction, Type Substitution Lemma 8, and Type Substitution Lemma 4
- SR4.1: the definition of substitution, Term Substitution Lemma 3, Type Substitution Lemma 8 (which applies because the right hypothesis ensures $C_R; C_\Delta \vdash_{\text{wf}} C_\Gamma$), and Type Substitution Lemma 6
- SR4.2: induction and Type Substitution Lemma 8 (which applies because the Typing Well-Formedness Lemma ensures $C_R; C_\Delta \vdash_{\text{wf}} C_\Gamma$)
- SR4.3–4: induction
- SR4.5: Type Substitution Lemma 6
- SR4.6–7: induction
- SR4.8: induction and Type Substitution Lemma 8 (which applies because the Typing Well-Formedness Lemma ensures $C_R; C_\Delta \vdash_{\text{wf}} C_\Gamma$)
- SR4.9: induction and Type Substitution Lemma 7
- SR4.10: induction and Term Substitution Lemma 1
- SR4.11: induction, Type Substitution Lemma 4, and Type Substitution Lemma 9 ensure we can derive a result that, given the Commuting Substitutions Lemma, is what we want.
- SR4.12: induction (applying the Commuting Substitutions Lemma to the result), Type Substitution Lemma 4, Type Substitution Lemma 9, and Type Substitution Lemma 4 again
- SR4.13: induction, Term Substitution Lemma 1, and Type Substitution Lemma 4
- SR4.14: induction, Type Substitution Lemma 6, and Type Substitution Lemma 4
- SR4.15: induction and Type Substitution Lemma 8 (which applies because the Typing Well-Formedness Lemma ensures $C_R; C_\Delta \vdash_{\text{wf}} C_\Gamma$)
- SR4.16: Type Substitution Lemma 6
- SR4.17: induction, Type Substitution Lemma 8, and Type Substitution Lemma 4
- SS4.1–5: induction
- SS4.6–7: induction and Type Substitution Lemma 4

$$\frac{\begin{array}{lll} S_G = i'_1{:}, H'_1 \ldots, i'_m{:}H'_m & R_G = i'_1, \ldots, i'_m & \gamma_G = \epsilon_1{<}i'_1, \ldots, \epsilon_m{<}i'_m \\ S_E = i_1{:}H_1, \ldots, i_j{:}H_j & R_E = i_1, \ldots, i_j & \epsilon = i_1 \cup \ldots \cup i_j \\ S_P = i_{j+1}{:}H_{j+1}, \ldots, i_n{:}H_n & R_P = i_{j+1}, \ldots, i_n & \gamma = i_1{<}i_2, i_2{<}i_3, \ldots, i_{n-1}{<}i_n \end{array} \quad R_G R_E R_P; \Gamma; \gamma\gamma_G \vdash_{\text{htyp}} S_G S_E S_P : \Gamma}{\vdash_{\text{hind}} S_G; S_E; S_P : R_G; R_E; R_P; \Gamma; \gamma\gamma_G; \epsilon}$$

$$\frac{\vdash_{\text{hind}} S_G; S_E; S_P : R_G; R_E; R_P; \Gamma; \gamma; \epsilon \quad R_G R_E R_P; \cdot; \Gamma; \gamma; \epsilon; \tau \vdash_{\text{styp}} s \quad R_P \vdash_{\text{spop}} s}{\vdash_{\text{sind}} S_G; S_E; S_P; \tau; s : R_G; R_E; R_P; \Gamma; \gamma}$$

$$\frac{\vdash_{\text{hind}} S_G; S_E; S_P : R_G; R_E; R_P; \Gamma; \gamma; \epsilon \quad R_G R_E R_P; \cdot; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} e : \tau \quad R_P \vdash_{\text{epop}} e}{\vdash_{\text{rind}} S_G; S_E; S_P; e : \tau; R_G; R_E; R_P; \Gamma; \gamma}$$

$$\frac{\vdash_{\text{hind}} S_G; S_E; S_P : R_G; R_E; R_P; \Gamma; \gamma; \epsilon \quad R_G R_E R_P; \cdot; \Gamma; \gamma; \epsilon \vdash_{\text{ltyp}} e : \tau, r \quad R_P \vdash_{\text{epop}} e}{\vdash_{\text{lind}} S_G; S_E; S_P; e : \tau, r; R_G; R_E; R_P; \Gamma; \gamma}$$

Figure B.1: Chapter 4 Safety-Proof Invariant

- SS4.8: Type Substitution Lemma 6, induction, and Type Substitution Lemma 4
- SS4.9: induction

Lemma B.11 (Return Preservation).

If $\vdash_{\text{ret}} s$ *and* $S_G; S; s \xrightarrow{\text{s}} S'_G; S'; s'$, *then* $\vdash_{\text{ret}} s'$.

Proof: The proof is by induction on the derivation of the dynamic step. It is very similar to the corresponding proof in Chapter 3, so we omit the details.

Lemma B.12 (Values Effectless).

1. *If* $R \vdash_{\text{epop}} v$ *or* $R \vdash_{\text{epop}} xp$, *then* $R = \cdot$.
2. *If* $R; \Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} v : \tau$ *and* $R; \Delta \vdash_{\text{wf}} \epsilon'$, *then* $R; \Delta; \Gamma; \gamma; \epsilon' \vdash_{\text{rtyp}} v : \tau$.
   If $R; \Delta; \Gamma; \gamma; \epsilon \vdash_{\text{ltyp}} xp : \tau, r$ *and* $R; \Delta \vdash_{\text{wf}} \epsilon'$, *then* $R; \Delta; \Gamma; \gamma; \epsilon' \vdash_{\text{ltyp}} xp : \tau, r$.

Proof: Both proofs are by induction on the structure of values. The noninductive cases show that the deallocation rules disallow a nonempty R and that the type-checking rules require nothing of ϵ except well-formedness.

Lemma B.13 (Access Control).

1. *If* $\vdash_{\text{hind}} S_G; S_E; S_P : R_G; R_E; R_P; \Gamma; \gamma; \epsilon$ *and* $\gamma; \epsilon \vdash_{\text{acc}} r$, *then* $r = \text{S}(i)$ *for some* $i \in \text{Dom}(S_E)$.

2. *If we further assume* $\Gamma(x) = (\tau, r)$*, then* $x \in \mathrm{Dom}(H)$ *for some* H *in* S_E*.*

Proof:

1. If $\gamma; \epsilon \vdash_{\mathrm{acc}} r$, then $\gamma \vdash_{\mathrm{eff}} \mathrm{regions}(r) \Leftarrow \epsilon$. Furthermore, the $\vdash_{\mathrm{hind}}$ hypotheses ensure ϵ describes only regions in S_E (and none of the form α). So it suffices to prove this stronger claim: If $\gamma \vdash_{\mathrm{eff}} \epsilon_1 \Leftarrow \epsilon_2$ and ϵ_2 describes only regions in S_E, then ϵ_1 describes only regions in S_E. The proof is by induction on the derivation of $\gamma \vdash_{\mathrm{eff}} \epsilon_1 \Leftarrow \epsilon_2$. The interesting case is when the last rule uses the fact that $\epsilon_1 < \epsilon_2$ is in γ. This case follows from the $\vdash_{\mathrm{hind}}$ hypotheses.

2. This lemma is a corollary to the previous one, given the Heap-Type Well-Formedness Lemma.

Lemma B.14 (Canonical Forms). *Suppose* $R; \cdot; \Gamma; \gamma; \epsilon \vdash_{\mathrm{rtyp}} v : \tau$*.*

1. *If* $\tau = \mathrm{int}$*, then* $v = i$ *for some* i*.*
2. *If* $\tau = \tau_0 \times \tau_1$*, then* $v = (v_0, v_1)$ *for some* v_0 *and* v_1*.*
3. *If* $\tau = \tau_0 \xrightarrow{\epsilon'} \tau_1$*, then* $v = (\tau_0, \rho\ x) \xrightarrow{\epsilon'} \tau_1\ s$ *for some* ρ*,* x*, and* s*.*
4. *If* $\tau = \tau' * r$*, then* $v = \&xp$ *for some* x *and* p*.*
5. *If* $\tau = \forall\alpha{:}\kappa[\gamma].\tau'$*, then* $v = \Lambda\alpha{:}\kappa[\gamma].f$ *for some* f*.*
6. *If* $\tau = \exists\alpha{:}\kappa[\gamma].\tau'$*, then* $v = \mathsf{pack}\ \tau'', v'\ \mathsf{as}\ \exists\alpha{:}\kappa[\gamma].\tau'$ *for some* τ'' *and* v'*.*
7. *If* $\tau = \mathrm{region}(r)$*, then* $r = \mathrm{S}(i)$ *and* $v = \mathsf{rgn}\ i$ *for some* i*.*

Proof: By inspection of the rules for $\vdash_{\mathrm{rtyp}}$ and the form of values

Lemma B.15 (Heap-Object Safety).

1. *If* $\vdash \mathrm{gettype}(\tau, p, \tau_0 \times \tau_1)$*, then* $\vdash \mathrm{gettype}(\tau, p0, \tau_0)$ *and* $\vdash \mathrm{gettype}(\tau, p1, \tau_1)$*.*
2. *If* $C \vdash_{\mathrm{rtyp}} v : \tau$*,* $\vdash \mathrm{gettype}(\tau, p, \tau')$*, and* $\mathrm{get}(v, p, v')$*, then* $C \vdash_{\mathrm{rtyp}} v' : \tau'$*.*
3. *If* $C \vdash_{\mathrm{rtyp}} v : \tau$*,* $\vdash \mathrm{gettype}(\tau, p, \tau')$*,* $C \vdash_{\mathrm{rtyp}} v' : \tau'$*, and* $\mathrm{set}(v, p, v', v'')$*, then* $C \vdash_{\mathrm{rtyp}} v'' : \tau$*.*
4. *If* $\vdash \mathrm{gettype}(\tau, p, \tau')$ *and* $C \vdash_{\mathrm{rtyp}} v : \tau$*, then there exists some* v' *such that* $\mathrm{get}(v, p, v')$*.*
5. *If* $\vdash \mathrm{gettype}(\tau, p, \tau')$ *and* $C \vdash_{\mathrm{rtyp}} v : \tau$*, then for all* v' *there exists some* v'' *such that* $\mathrm{set}(v, p, v', v'')$*.*

6. *If* $\cdot \vdash_{\text{epop}} v$ *and* $\text{get}(v, p, v')$*, then* $\cdot \vdash_{\text{epop}} v'$.

7. *If* $\cdot \vdash_{\text{epop}} v$, $\cdot \vdash_{\text{epop}} v'$, *and* $\text{set}(v, p, v', v'')$*, then* $\cdot \vdash_{\text{epop}} v''$.

Proof: In all cases, the proof is by induction on the length of p.

1. If $p = \cdot$, the result is immediate, else the induction hypothesis suffices.

2. If $p = \cdot$, the result is immediate, else the induction hypothesis suffices.

3. If $p = \cdot$, the result is immediate given $C \vdash_{\text{rtyp}} v' : \tau'$, else the induction hypothesis, inversion of the derivation of $C \vdash_{\text{rtyp}} v : \tau$, and rule SR4.7 suffice.

4. If $p = \cdot$, let $v' = v$. Else the induction hypothesis and the Canonical Forms Lemma suffice.

5. If $p = \cdot$, let $v'' = v'$. Else the induction hypothesis and the Canonical Forms Lemma suffice.

6. If $p = \cdot$, the result is immediate, else the induction hypothesis and the definition of $\vdash_{\text{epop}}$ suffices.

7. If $p = \cdot$, the result is immediate, else the induction hypothesis and the definition of $\vdash_{\text{epop}}$ suffices.

Lemma B.16 (Subtyping Preservation).

1. *If* $C \vdash_{\text{rtyp}} \&xp : \tau * r$*, then* $C \vdash_{\text{ltyp}} xp : \tau, r$.

2. *If* $C \vdash_{\text{ltyp}} xp : \tau, r$ *and* $C_\gamma; C_\epsilon \vdash_{\text{acc}} r$*, then* $C \vdash_{\text{rtyp}} xp : \tau$.

3. *If* $C \vdash_{\text{rtyp}} \&xp : \tau * r$ *and* $C_\gamma; C_\epsilon \vdash_{\text{acc}} r$*, then* $C \vdash_{\text{rtyp}} xp : \tau$.

4. *If* $C \vdash_{\text{ltyp}} xp : \tau_0 \times \tau_1, r$*, then* $C \vdash_{\text{ltyp}} xp0 : \tau_0, r$ *and* $C \vdash_{\text{ltyp}} xp1 : \tau_1, r$.

Proof:

1. By induction on the derivation of $C \vdash_{\text{rtyp}} \&xp : \tau * r$: If the last rule is SR4.6, its hypothesis suffices. Else the last rule is SR4.17, so there is an r' such that $C \vdash_{\text{rtyp}} \&xp : \tau * r'$, $C_\gamma; \text{regions}(r) \vdash_{\text{acc}} r'$, and $C_R; C_\Delta \vdash_{\text{k}} r : \texttt{R}$. By induction $C \vdash_{\text{ltyp}} xp : \tau, r'$, so SL4.5 ensures the desired result.

2. By induction on the derivation of $C \vdash_{\text{ltyp}} xp : \tau, r$: If the last rule is SL4.1, then its hypotheses, $C_\gamma; C_\epsilon \vdash_{\text{acc}} r$, and SR4.1 ensure the desired result. Else the last rule is SL4.5, so there is an r' such that $C \vdash_{\text{ltyp}} xp : \tau, r'$ and $C_\gamma; \text{regions}(r) \vdash_{\text{acc}} r'$. Given $C_\gamma; C_\epsilon \vdash_{\text{acc}} r$ and $C_\gamma; \text{regions}(r) \vdash_{\text{acc}} r'$, we know $C_\gamma \vdash_{\text{eff}} \text{regions}(r) \Leftarrow$

C_ϵ and $C_\gamma \vdash_{\text{eff}} \text{regions}(r') \Leftarrow \text{regions}(r)$. So we can derive $C_\gamma \vdash_{\text{eff}} \text{regions}(r') \Leftarrow C_\epsilon$; hence $C_\gamma; C_\epsilon \vdash_{\text{acc}} r'$. (The Typing Well-Formedness Lemma and the Type Canonical Forms Lemma ensure $\vdash_{\text{acc}}$ applies.) Hence the induction hypothesis ensures $C \vdash_{\text{rtyp}} xp : \tau$.

3. This lemma is a corollary of the previous two lemmas.

4. By induction on the derivation of $C \vdash_{\text{ltyp}} xp : \tau_0 \times \tau_1, r$: If the last rule is SL4.1, inversion and Heap-Object Safety Lemma 1 suffice. Else the last rule is SL4.5, and the result follows from induction.

Lemma B.17 (New-Region Preservation).
If $\epsilon = i_1 \cup \ldots \cup i_n$ *and* $\gamma = i_1{<}i_2, \ldots, i_{n-1}{<}i_n$, *then* $\gamma, i_n{<}i \vdash_{\text{eff}} \gamma, \epsilon{<}i$.

Proof: By induction on n: If $n = 0$, our obligation is $\cdot \vdash_{\text{eff}} \emptyset{<}i$, which we can prove by rewriting i as $\emptyset \cup i$ and showing $\cdot \vdash_{\text{eff}} \emptyset \Leftarrow \emptyset$. For $n > 0$, let $\epsilon' = i_1 \cup \ldots \cup i_{n-1}$ and $\gamma' = i_1{<}i_2, \ldots, i_{n-2}{<}i_{n-1}$. Invoking the induction hypothesis with i_n for i ensures $\gamma \vdash_{\text{eff}} \gamma', \epsilon'{<}i_n$. The Context Weakening Lemma ensures $\gamma, i_n{<}i \vdash_{\text{eff}} \gamma', \epsilon'{<}i_n$, so we can derive $\gamma, i_n{<}i \vdash_{\text{eff}} \gamma', \epsilon'{<}i_n, i_n{<}i$. So the Context Weakening Lemma ensures it suffices to show $\gamma', \epsilon'{<}i_n, i_n{<}i \vdash_{\text{eff}} \gamma, \epsilon{<}i$, for which it suffices to show $\gamma', \epsilon'{<}i_n, i_n{<}i \vdash_{\text{eff}} \gamma'$ (trivial), $\gamma', \epsilon'{<}i_n, i_n{<}i \vdash_{\text{eff}} i_{n-1}{<}i_n$ (trivial because $\epsilon' = i_{n-1} \cup \epsilon''$ for some ϵ''), and $\gamma', \epsilon'{<}i_n, i_n{<}i \vdash_{\text{eff}} \epsilon{<}i$ (follows from $\epsilon'{<}i_n$ and $i_n{<}i$ because $\epsilon = \epsilon' \cup i_n$).

Lemma B.18 (Type and Pop Preservation). *Suppose:*

1. $S_G; S_E S_P; s \xrightarrow{\text{s}} S'_G; S'; s'$
   (respectively, $S_G; S_E S_P; e \xrightarrow{\text{r}} S'_G; S'; e'$*)*
   (respectively, $S_G; S_E S_P; e \xrightarrow{\text{l}} S'_G; S'; e'$*)*

2. $\vdash_{\text{sind}} S_G; S_E; S_P; \tau; s : R_G; R_E; R_P; \Gamma; \gamma$
   (respectively, $\vdash_{\text{rind}} S_G; S_E; S_P; e : \tau; R_G; R_E; R_P; \Gamma; \gamma$*)*
   (respectively, $\vdash_{\text{lind}} S_G; S_E; S_P; e : \tau, r; R_G; R_E; R_P; \Gamma; \gamma$*)*

Then there exist S'_G, S'_E, S'_P, R'_G, R'_P, Γ', *and* γ' *such that:*

1. $S' = S'_E S'_P$
2. $R_G R_P \subseteq R'_G R'_P$
3. $\Gamma' = \Gamma\Gamma''$ *for some* Γ''
4. $\gamma' \vdash_{\text{eff}} \gamma$

5. $\vdash_{\text{sind}} S'_G; S'_E; S'_P; \tau; s' : R'_G; R_E; R'_P; \Gamma'; \gamma'$
   (respectively, $\vdash_{\text{rind}} S'_G; S'_E; S'_P; e' : \tau; R'_G; R_E; R'_P; \Gamma'; \gamma'$*)*
   (respectively, $\vdash_{\text{lind}} S'_G; S'_E; S'_P; e' : \tau, r; R'_G; R_E; R'_P; \Gamma'; \gamma'$*)*

Proof: The proofs are by simultaneous induction on the typing derivations implied by the $\vdash_{\text{sind}}$, $\vdash_{\text{rind}}$, and $\vdash_{\text{lind}}$ assumptions, proceeding by cases on the last rule used. (Subtyping makes this technique easier than induction on the dynamic derivations.) In each case, let $C = R_G R_E R_P; \cdot; \Gamma; \gamma; \epsilon$ and $C' = R'_G R_E R'_P; \cdot; \Gamma'; \gamma'; \epsilon$.

Two situations arise often in the proof, so we sketch the structure of the argument for these situations. First, if the dynamic step does not change the heap, (e.g., $S_G; S; s \xrightarrow{s} S_G; S; s'$), then we say the situation is *local*. Letting $S'_E = S_E$, $S'_P = S_P$, $R'_G = R_G$, $R'_P = R_P$, $\Gamma' = \Gamma$, and $\gamma' = \gamma$, it suffices to show $C; \tau \vdash_{\text{styp}} s'$ and $R_P \vdash_{\text{spop}} s'$ (respectively, $C \vdash_{\text{rtyp}} e' : \tau$ and $R_P \vdash_{\text{epop}} e'$) (respectively, $C \vdash_{\text{ltyp}} e' : \tau, r$ and $R_P \vdash_{\text{epop}} e'$).

Second, all arguments that use induction follow a similar form. We say the situation is *inductive*. To invoke the induction hypothesis, we use the $\vdash_{\text{hind}}$ assumption without change and we use inversion on the type-checking and deallocation assumptions to conclude the type-checking and deallocation facts induction requires. Invoking the induction hypothesis provides $\vdash_{\text{hind}} S'_G; S'_E; S'_P : R'_G R_E; R'_P; \Gamma'; \gamma'; \epsilon$ for S'_E, S'_P, R'_G, R'_P, Γ', and γ' satisfying conclusions 1–4. It also provides type-checking and deallocation results that we use to derive the type-checking and deallocation results we need. Conclusions 1–4 let us apply various weakening lemmas to establish other hypotheses necessary for the type-checking result. Given the $\vdash_{\text{hind}}$ result from the induction and the type-checking and deallocation results established for each situation, conclusion 5 follows.

- SL4.1: This case holds vacuously because no dynamic rule applies.

- SL4.2: Let $e = *e_1$. By inversion $C \vdash_{\text{rtyp}} e_1 : \tau * r$ and $R_P \vdash_{\text{epop}} e_1$. Only DL4.2 or DL4.3 applies. For DL4.2, $e_1 = \&xp$ and the situation is local. Subtyping Preservation Lemma 1 ensures $C \vdash_{\text{ltyp}} xp : \tau, r$. Inversion ensures $R_P \vdash_{\text{epop}} xp$. For DL4.3, the situation is inductive and e_1 becomes e'_1, so $C' \vdash_{\text{rtyp}} e'_1 : \tau * r$ and $R'_P \vdash_{\text{epop}} e'_1$. So $C' \vdash_{\text{ltyp}} *e'_1 : \tau, r$ and $R'_P \vdash_{\text{epop}} *e'_1$.

- SL4.3–4: Let $e = e_1.i$ and $\tau = \tau_i$. By inversion $C \vdash_{\text{rtyp}} e_1 : \tau_0 \times \tau_1$ and $R_P \vdash_{\text{epop}} e_1$. Only DL4.1 or DL4.4 applies. For DL4.1, $e_1 = xp$ and the situation is local. Subtyping Preservation Lemma 4 ensures $C \vdash_{\text{ltyp}} xpi : \tau_i, r$. Inspection ensures $R_P \vdash_{\text{epop}} xpi$. For DL4.4, the situation is inductive and e_1 becomes e'_1, so $C' \vdash_{\text{ltyp}} e'_1 : \tau_0 \times \tau_1, r$ and $R'_P \vdash_{\text{epop}} e'_1$. So $C' \vdash_{\text{ltyp}} e'_1.i : \tau_i, r$ and $R'_P \vdash_{\text{epop}} e'_1.i$.

- SL4.5: Inversion ensures $C \vdash_{\text{ltyp}} e : \tau, r'$, $\gamma; \text{regions}(r) \vdash_{\text{acc}} r'$, and $C_R; \cdot \vdash_{\text{k}} r : \texttt{R}$. The situation is inductive (using r' for r so the $\vdash_{\text{epop}}$ hypothesis applies unchanged). So $C' \vdash_{\text{ltyp}} e' : \tau, r'$ and $R'_P \vdash_{\text{epop}} e'$. Context Weakening Lemmas 6 and 3 ensure $\gamma'; \text{regions}(r) \vdash_{\text{acc}} r'$ and $C'_R; \cdot \vdash_{\text{k}} r : \texttt{R}$, so $C' \vdash_{\text{ltyp}} e' : \tau, r$.

- SR4.1: Inversion ensures $\Gamma(x) = (\tau', r)$, $\vdash \text{gettype}(\tau', p, \tau)$, $\vdash_{\text{wf}} C$, and $R_P \vdash_{\text{epop}} xp$ (and therefore $R_P = \cdot$). Only DR4.1 applies and the situation is local. The Heap-Type Well-Formedness Lemma ensures $\cdot \vdash_{\text{epop}} H(x)$ and $C_R; \cdot; \Gamma; \gamma; \emptyset \vdash_{\text{rtyp}} H(x) : \tau'$, which by the Term Weakening Lemma ensures $C \vdash_{\text{rtyp}} H(x) : \tau'$. So Heap-Object Safety Lemmas 6 and 2 ensure $\cdot \vdash_{\text{epop}} v$ and $C \vdash_{\text{rtyp}} v : \tau$.

- SR4.2: Let $e = *e_1$. By inversion $C \vdash_{\text{rtyp}} e_1 : \tau*r$, $\gamma; \epsilon \vdash_{\text{acc}} r$, and $R_P \vdash_{\text{epop}} e_1$. Only DR4.3 or DR4.11 applies. For DR4.3, $e_1 = \&xp$ and the situation is local. Subtyping Preservation Lemma 3 ensures $C \vdash_{\text{rtyp}} xp : \tau$. Inversion ensures $R_P \vdash_{\text{epop}} xp$. For DR4.11, the situation is inductive and e_1 becomes e'_1, so $C' \vdash_{\text{rtyp}} e'_1 : \tau*r$ and $R'_P \vdash_{\text{epop}} e'_1$. Context Weakening Lemma 6 ensures $\gamma'; \epsilon \vdash_{\text{acc}} r$. So $C' \vdash_{\text{rtyp}} *e'_1 : \tau$ and $R'_P \vdash_{\text{epop}} *e'_1$.

- SR4.3–4: Let $e = e_1.i$ and $\tau = \tau_i$. By inversion $C \vdash_{\text{rtyp}} e_1 : \tau_0 \times \tau_1$ and $R_P \vdash_{\text{epop}} e_1$. Only DR4.4 or DR4.11 applies. For DR4.4, $e_1 = (v_0, v_1)$ and the situation is local. Inversion and the Values Effectless Lemma ensure $C \vdash_{\text{rtyp}} v_i : \tau_i$ and $R_P \vdash_{\text{epop}} v_i$ (because $R_P = \cdot$). For DR4.11, the situation is inductive and e_1 becomes e'_1, so $C' \vdash_{\text{rtyp}} e'_1 : \tau_0 \times \tau_1$ and $R'_P \vdash_{\text{epop}} e'_1$. So $C' \vdash_{\text{rtyp}} e'_1.i : \tau_i$ and $R'_P \vdash_{\text{epop}} e'_1.i$.

- SR4.5: This case holds vacuously because no dynamic rule applies.

- SR4.6: Let $e = \&e_1$ and $\tau = \tau_1*r$. By inversion $C \vdash_{\text{ltyp}} e_1 : \tau_1, r$ and $R_P \vdash_{\text{epop}} e_1$. Only DR4.10 applies. The situation is inductive with e_1 becoming e'_1. So $C' \vdash_{\text{ltyp}} e'_1 : \tau_1, r$ and $R_P \vdash_{\text{epop}} e'_1$. So $C' \vdash_{\text{rtyp}} \&e'_1 : \tau_1*r$ and $R_P \vdash_{\text{epop}} \&e'_1$.

- SR4.7: Let $e = (e_0, e_1)$ and $\tau = \tau_0 \times \tau_1$. By inversion $C \vdash_{\text{rtyp}} e_0 : \tau_0$ and $C \vdash_{\text{rtyp}} e_1 : \tau_1$. Only DR4.11 applies; either e_0 is a value or not. For e_0 a value, the situation is inductive and e_1 becomes e'_1. By inversion $\cdot \vdash_{\text{epop}} e_0$ and $R_P \vdash_{\text{epop}} e_1$. (Inversion of $R_P \vdash_{\text{epop}} e$ could provide $R_P \vdash_{\text{epop}} e_0$ and $\cdot \vdash_{\text{epop}} e_1$, but then the Values Effectless Lemma ensures $R_P = \cdot$.) By induction $C' \vdash_{\text{rtyp}} e'_1 : \tau_1$ and $R'_P \vdash_{\text{epop}} e'_1$. By the Term Weakening Lemma $C' \vdash_{\text{rtyp}} e_0 : \tau_0$. So $C' \vdash_{\text{rtyp}} (e_0, e'_1) : \tau_0 \times \tau_1$ and $R'_P \vdash_{\text{epop}} (e_0, e'_1)$. For e_0 not a value, the situation is inductive and e_0 becomes e'_0. By inversion, $R_P \vdash_{\text{epop}} e_0$ and $\cdot \vdash_{\text{epop}} e_1$. By induction $C' \vdash_{\text{rtyp}} e'_0 : \tau_0$ and $R'_P \vdash_{\text{epop}} e'_0$. By the Term Weakening Lemma, $C' \vdash_{\text{rtyp}} e_1 : \tau_1$. So $C' \vdash_{\text{rtyp}} (e'_0, e_1) : \tau_0 \times \tau_1$ and $R'_P \vdash_{\text{epop}} (e'_0, e_1)$.

- SR4.8: Let $e = (e_1{=}e_2)$. By inversion $C \vdash_{\text{ltyp}} e_1 : \tau, r$, $C \vdash_{\text{rtyp}} e_2 : \tau$, and $\gamma; \epsilon \vdash_{\text{acc}} r$. Only DR4.2, DR4.10, or DR4.11 applies. For DR4.2, let $e_1 = xp$ and $e_2 = v$. By inversion and the Values Effectless Lemma, $R_P = \cdot$, $\cdot \vdash_{\text{epop}} v$, $C_\Gamma(x) = (\tau', r)$, $\vdash \text{gettype}(\tau', p, \tau)$, and $C_R; \cdot; \Gamma; \gamma; \emptyset \vdash_{\text{rtyp}} v : \tau$. The Heap-Type Well-Formedness Lemma ensures location x holds some v' such that $C_R; \cdot; \Gamma; \gamma; \emptyset \vdash_{\text{rtyp}} v' : \tau'$ and $\cdot \vdash_{\text{epop}} v'$. So given $\text{set}(v', p, v, v'')$, Heap-Object Safety Lemmas 3 and 7 ensure $C_R; \cdot; \Gamma; \gamma; \emptyset \vdash_{\text{rtyp}} v'' : \tau'$ and $\cdot \vdash_{\text{epop}} v''$. So letting S'_E and S'_P be S_E and S_P except x holds v'', a trivial induction on the $\vdash_{\text{hind}}$ assumption $C_R; \Gamma; \gamma \vdash_{\text{htyp}} S_G S_E S_P : \Gamma$ shows $C_R; \Gamma; \gamma \vdash_{\text{htyp}} S_G S'_E S'_P : \Gamma$. (In fact, $\gamma; \epsilon \vdash_{\text{acc}} r$ ensures x is in S_E.) So $\vdash_{\text{hind}} S_G; S'_E; S'_P; R_G; R_E; R_P; \Gamma; \gamma; \epsilon$. Because $e' = v$, $C \vdash_{\text{rtyp}} v : \tau$, and $R_P \vdash_{\text{epop}} v$, all the conclusions follow. For DR4.10, the situation is inductive and e_1 becomes e'_1. By inversion, $R_P \vdash_{\text{epop}} e_1$ and $\cdot \vdash_{\text{epop}} e_2$. By induction $C' \vdash_{\text{ltyp}} e'_1 : \tau, r$ and $R'_P \vdash_{\text{epop}} e'_1$. By the Term Weakening Lemma, $C' \vdash_{\text{rtyp}} e_2 : \tau$. By the Context Weakening Lemma $\gamma'; \epsilon \vdash_{\text{acc}} r$. So $C' \vdash_{\text{rtyp}} e'_1{=}e_2 : \tau$ and $R'_P \vdash_{\text{epop}} e'_1{=}e_2$. For DR4.11, the situation is inductive and e_2 becomes e'_2. By inversion, $\cdot \vdash_{\text{epop}} xp$ and $R_P \vdash_{\text{epop}} e_2$. (Inversion of $R_P \vdash_{\text{epop}} e$ could provide $R_P \vdash_{\text{epop}} e_1$ and $\cdot \vdash_{\text{epop}} e_2$, but then the Values Effectless Lemma ensures $R_P = \cdot$.) By induction $C' \vdash_{\text{rtyp}} e'_2 : \tau$ and $R'_P \vdash_{\text{epop}} e'_2$. By the Term Weakening Lemma, $C' \vdash_{\text{ltyp}} e_1 : \tau, r$. By the Context Weakening Lemma $\gamma'; \epsilon \vdash_{\text{acc}} r$. So $C' \vdash_{\text{rtyp}} e_1{=}e'_2 : \tau$ and $R'_P \vdash_{\text{epop}} e_1{=}e'_2$.

- SR4.9: Let $e = e_1(e_2)$. By inversion $C \vdash_{\text{rtyp}} e_1 : \tau' \xrightarrow{\epsilon'} \tau$, $C \vdash_{\text{rtyp}} e_2 : \tau'$, and $\gamma \vdash_{\text{eff}} \epsilon' \Leftarrow \epsilon$. Only DR4.5 or DR4.11 applies. For DR4.5, the situation is local and e becomes $\mathsf{call}\ (\mathsf{let}\ \rho, x = v;\ s)$. Let $e_1 = (\tau', \rho\ x) \xrightarrow{\epsilon'} \tau\ s$ and $e_2 = v$. By inversion and the Values Effectless Lemma, $R_P = \cdot$, $\cdot \vdash_{\text{epop}} v$, $\cdot \vdash_{\text{spop}} s$, $\vdash_{\text{ret}} s$, and $C_R; \rho{:}\mathtt{R}; \Gamma, x{:}\tau'; \gamma, \epsilon'{<}\rho; \epsilon' \cup \rho; \tau \vdash_{\text{styp}} s$. By the Context Weakening Lemma, $C_R; \rho{:}\mathtt{R}; \Gamma, x{:}\tau'; \gamma, \epsilon{<}\rho; \epsilon \cup \rho; \tau \vdash_{\text{styp}} s$. So we can derive $C \vdash_{\text{rtyp}} \mathsf{call}\ (\mathsf{let}\ \rho, x = v;\ s) : \tau$ and $\cdot \vdash_{\text{epop}} \mathsf{call}\ (\mathsf{let}\ \rho, x = v;\ s)$. For DR4.11, the situation is inductive. The argument is like the argument for SR4.7 (using e_1 for e_0 and e_2 for e_1) with the addition that the Context Weakening Lemma ensures $\gamma' \vdash_{\text{eff}} \epsilon' \Leftarrow \epsilon$.

- SR4.10: Let $e = \mathsf{call}\ s$. By inversion $C; \tau \vdash_{\text{styp}} s$, $\vdash_{\text{ret}} s$ and $R_P \vdash_{\text{spop}} s$. Only DR4.6 and DR4.9 apply. For DR4.6, the situation is local, $s = \mathsf{return}\ v$ and e becomes v. Inversion ensures $C \vdash_{\text{rtyp}} v : \tau$ and $R_P \vdash_{\text{epop}} v$. For DR4.9, the situation is inductive and s becomes s', so $C'; \tau \vdash_{\text{styp}} s'$ and $R'_P \vdash_{\text{spop}} s'$. The Return Preservation Lemma ensures $\vdash_{\text{ret}} s'$, so $C' \vdash_{\text{rtyp}} \mathsf{call}\ s' : \tau$ and $R'_P \vdash_{\text{epop}} \mathsf{call}\ s'$.

- SR4.11: Let $e = e_1[\tau_2]$ and $\tau = \tau_1[\tau_2/\alpha]$. By inversion $C \vdash_{\text{rtyp}} e : \forall\alpha{:}\kappa[\gamma_1].\tau_1$,

$C_R;\cdot \vdash_{\text{k}} \tau_2 : \kappa$, $\gamma \vdash_{\text{eff}} \gamma_1[\tau_2/\alpha]$, and $R_P \vdash_{\text{epop}} e_1$. Only DR4.7 or DR4.11 applies. For DR4.7, the situation is local. Inversion ensures $e_1 = \Lambda\alpha{:}\kappa[\gamma_1].f$, $C_R;\alpha{:}\kappa;\Gamma;\gamma\gamma_1;\epsilon \vdash_{\text{rtyp}} f : \tau_1$, $\vdash_{\text{wf}} C$, and $R_P = \cdot$. The Substitution Lemma, Useless Substitution Lemma, and $\vdash_{\text{wf}} C$ ensure $C_R;\cdot;\Gamma;\gamma(\gamma_1[\tau_2/\alpha]);\epsilon \vdash_{\text{rtyp}} f[\tau_2/\alpha] : \tau_1[\tau_2/\alpha]$ and $\cdot \vdash_{\text{epop}} f[\tau_2/\alpha]$. The Context Weakening Lemma and $\gamma \vdash_{\text{eff}} \gamma_1[\tau_2/\alpha]$ ensure $C \vdash_{\text{rtyp}} f[\tau_2/\alpha] : \tau_1[\tau_2/\alpha]$. For DR4.11, the situation is inductive and e_1 becomes e_1', so $C' \vdash_{\text{rtyp}} e_1' : \forall\alpha{:}\kappa[\gamma_1].\tau_1$ and $R_P' \vdash_{\text{epop}} e_1'$. So $C' \vdash_{\text{rtyp}} e_1'[\tau_2] : \tau_1[\tau_2/\alpha]$ and $R_P' \vdash_{\text{epop}} e_1'[\tau_2]$.

- SR4.12: Let $e = \mathsf{pack}\ \tau_2, e_1\ \mathsf{as}\ \exists\alpha{:}\kappa[\gamma_1].\tau_1$ and $\tau = \exists\alpha{:}\kappa[\gamma_1].\tau_1$. Only DR4.11 applies. The situation is inductive and e_1 becomes e_1'. By inversion, $C \vdash_{\text{rtyp}} e_1 : \tau_1[\tau_2/\alpha]$, $C_R;\cdot \vdash_{\text{k}} \tau_2 : \kappa$, $\gamma \vdash_{\text{eff}} \gamma_1[\tau_2/\alpha]$, $C_R;\cdot \vdash_{\text{k}} \tau : \mathtt{A}$, and $R_P \vdash_{\text{epop}} e_1$. By induction, $C' \vdash_{\text{rtyp}} e_1' : \tau_1[\tau_2/\alpha]$ and $R_P' \vdash_{\text{epop}} e_1'$. By the Context Weakening Lemma, $C_R';\cdot \vdash_{\text{k}} \tau_2 : \kappa$, $\gamma' \vdash_{\text{eff}} \gamma_1[\tau_2/\alpha]$ and $C_R';\cdot \vdash_{\text{k}} \tau : \mathtt{A}$. So $C' \vdash_{\text{rtyp}} \mathsf{pack}\ \tau_2, e_1'\ \mathsf{as}\ \exists\alpha{:}\kappa[\gamma_1].\tau_1 : \tau$ and $R_P' \vdash_{\text{epop}} \mathsf{pack}\ \tau_2, e_1'\ \mathsf{as}\ \exists\alpha{:}\kappa[\gamma_1].\tau_1$.

- SR4.13–14: These cases hold vacuously because no dynamic rule applies.

- SR4.15: Let $e = \mathsf{rnew}\ e_1\ e_2$ and $\tau = \tau'{*}r$. By inversion $C \vdash_{\text{rtyp}} e_1 : \text{region}(r)$, $C \vdash_{\text{rtyp}} e_2 : \tau'$, and $\gamma;\epsilon \vdash_{\text{acc}} r$. Only DR4.8 or DR4.11 applies. For DR4.8, $e_1 = \mathsf{rgn}\ i$ and $e_2 = v$. Inversion ensures $r = \mathrm{S}(i)$, $R_P = \cdot$, and $\cdot \vdash_{\text{epop}} v$. The Values Effectless Lemma ensures $C_R;\cdot;\Gamma;\gamma;\emptyset \vdash_{\text{rtyp}} v : \tau'$. So given the $\vdash_{\text{hind}}$ assumption, a trivial induction on the $\vdash_{\text{htyp}}$ derivation shows $C_R;\Gamma,x{:}(\tau',\mathrm{S}(i));\gamma \vdash_{\text{htyp}} S_G S_E' S_P' : \Gamma, x{:}(\tau',\mathrm{S}(i))$ where S_E' and S_P' are like S_E and S_P except i now has a location x holding v. (In fact, $\gamma;\epsilon \vdash_{\text{acc}} r$ ensures x is in S_E'.) So $\vdash_{\text{hind}} S_G; S_E'; S_P' : R_G; R_E; R_P; \Gamma, x{:}(\tau',\mathrm{S}(i)); \gamma; \epsilon$. We can derive $C_R;\cdot;\Gamma,x{:}(\tau',\mathrm{S}(i));\gamma;\epsilon \vdash_{\text{rtyp}} \&x\cdot : \tau'{*}\mathrm{S}(i)$ and $R_P \vdash_{\text{epop}} \&x\cdot$, so we can conclude the $\vdash_{\text{rind}}$ fact conclusion 5 requires. The other conclusions follow because $\Gamma, x{:}(\tau',\mathrm{S}(i))$ extends Γ and the rest of the context is unchanged.

  For DR4.11, the situation is inductive. The argument is like the argument for SR4.7 (using e_1 for e_0 and e_2 for e_1) with the addition that the Context Weakening Lemma ensures $\gamma';\epsilon \vdash_{\text{acc}} r$.

- SR4.16: This case holds vacuously because no dynamic rule applies.

- SR4.17: Let $\tau = \tau_1{*}r$. Inversion ensures $C \vdash_{\text{rtyp}} e : \tau_1{*}r'$, $\gamma;\text{regions}(r) \vdash_{\text{acc}} r'$, and $C_R;\cdot \vdash_{\text{k}} r : \mathtt{R}$. The situation is inductive (using r' for r so the $\vdash_{\text{epop}}$ hypothesis applies unchanged). So $C' \vdash_{\text{rtyp}} e' : \tau_1{*}r'$ and $R_P' \vdash_{\text{epop}} e'$. Context Weakening Lemmas 6 and 3 ensure $\gamma';\text{regions}(r) \vdash_{\text{acc}} r'$ and $C_R';\cdot \vdash_{\text{k}} r : \mathtt{R}$, so $C' \vdash_{\text{rtyp}} e' : \tau_1{*}r$.

- SS4.1–2: In both cases, only DS4.11 applies and the situation is inductive. By inversion $C \vdash_{\text{rtyp}} e : \tau'$ (for SS4.2 $\tau' = \tau$) and $R_P \vdash_{\text{epop}} e$. By induction $C' \vdash_{\text{rtyp}} e' : \tau'$ and $R'_P \vdash_{\text{epop}} e'$. So $C'; \tau \vdash_{\text{styp}} s'$ and $R'_P \vdash_{\text{spop}} s'$.

- SS4.3: Let $s = s_1; s_2$. By inversion $C; \tau \vdash_{\text{styp}} s_1$ and $C; \tau \vdash_{\text{styp}} s_2$. Only DS4.2, DS4.3, or DS4.12 applies. For DS4.2, the situation is local, $s_1 = v$, and s becomes s_2. By inversion $R_P \vdash_{\text{epop}} v$ (so by the Values Effectless Lemma $R_P = \cdot$) and $\cdot \vdash_{\text{spop}} s_2$. So $R_P \vdash_{\text{spop}} s_2$. For DS4.3, the situation is local and s becomes s_1. By inversion $R_P \vdash_{\text{spop}} s_1$. For DS4.12, the situation is inductive and s_1 becomes s'_1. By inversion $R_P \vdash_{\text{spop}} s_1$ and $\cdot \vdash_{\text{spop}} s_2$. By induction $C'; \tau \vdash_{\text{styp}} s'_1$ and $R'_P \vdash_{\text{spop}} s'_1$. By the Term Weakening Lemma, $C'; \tau \vdash_{\text{styp}} s_2$. So $C'; \tau \vdash_{\text{styp}} s'_1; s_2$ and $R'_P \vdash_{\text{spop}} s'_1; s_2$.

- SS4.4: Let $s = \mathsf{while}\ e\ s_1$. Only DS4.6 applies and the situation is local. By inversion $C \vdash_{\text{rtyp}} e : \text{int}$, $C; \tau \vdash_{\text{styp}} s_1$, $\cdot \vdash_{\text{epop}} e$, $\cdot \vdash_{\text{spop}} s_1$, and $R_P = \cdot$. Trivially, $C \vdash_{\text{rtyp}} 0 : \text{int}$ and $\cdot \vdash_{\text{epop}} 0$. So $C; \tau \vdash_{\text{styp}} \mathsf{if}\ e\ (s_1; \mathsf{while}\ e\ s_1)\ 0$ and $\cdot \vdash_{\text{spop}} \mathsf{if}\ e\ (s_1; \mathsf{while}\ e\ s_1)\ 0$.

- SS4.5: Let $s = \mathsf{if}\ e\ s_1\ s_2$. By inversion $C \vdash_{\text{rtyp}} e : \text{int}$, $C; \tau \vdash_{\text{styp}} s_1$, $C; \tau \vdash_{\text{styp}} s_2$, $R_P \vdash_{\text{epop}} e$, $\cdot \vdash_{\text{spop}} s_1$, and $\cdot \vdash_{\text{spop}} s_2$. Only DS4.4, DS4.5, or DS4.11 applies. For DS4.4, the situation is local, $e = 0$, and s becomes s_1. By the Values Effectless Lemma $R_P = \cdot$, so $R_P \vdash_{\text{spop}} s_1$. The proof for DS4.5 is analogous, using i and s_2 for 0 and s_1. For DS4.11, the situation is inductive and e becomes e'. By induction $C' \vdash_{\text{rtyp}} e' : \text{int}$ and $R'_P \vdash_{\text{epop}} e'$. By the Term Weakening Lemma, $C'; \tau \vdash_{\text{styp}} s_1$ and $C'; \tau \vdash_{\text{styp}} s_2$. So $C'; \tau \vdash_{\text{styp}} \mathsf{if}\ e'\ s_1\ s_2$ and $R'_P \vdash_{\text{spop}} \mathsf{if}\ e'\ s_1\ s_2$.

- SS4.6: Let $s = \mathsf{let}\ \rho, x = e;\ s_1$. Only DS4.1 and DS4.11 apply. For DS4.1, the argument is analogous to case SS4.8 below, so we explain only the differences: We use e (which is a value v) in place of $\mathsf{rgn}\ i$ and the type of e (τ') in place of region(S(i)). To conclude $R_P = \cdot$, we use the Values Effectless Lemma and $R_P \vdash_{\text{epop}} e$. We also need the Values Effectless Lemma to show that e is well-typed in the heap (under capability $\emptyset$). For DS4.11, the situation is inductive and e becomes e'. By inversion, $C \vdash_{\text{rtyp}} e : \tau'$, $C_R; \rho{:}\mathtt{R}; \Gamma, x{:}(\tau', \rho); \gamma, \epsilon{<}\rho; \epsilon \cup \rho; \tau \vdash_{\text{styp}} s_1$, $C_R; \cdot \vdash_{\text{k}} \tau : \mathtt{A}$, $R_P \vdash_{\text{epop}} e$, and $\cdot \vdash_{\text{epop}} s_1$. By induction $C' \vdash_{\text{rtyp}} e' : \tau'$ and $R'_P \vdash_{\text{epop}} e'$ By the Term Weakening Lemma, $C'_R; \rho{:}\mathtt{R}; \Gamma', x{:}(\tau', \rho); \gamma', \epsilon{<}\rho; \epsilon \cup \rho; \tau \vdash_{\text{styp}} s_1$. By the Context Weakening Lemma, $C'_R; \cdot \vdash_{\text{k}} \tau : \mathtt{A}$. So $C'; \tau \vdash_{\text{styp}} \mathsf{let}\ \rho, x = e';\ s_1$ and $R'_P \vdash_{\text{spop}} \mathsf{let}\ \rho, x = e';\ s_1$.

- SS4.7: Let $s = \mathsf{open}\ e\ \mathsf{as}\ \rho, \alpha, x;\ s_1$. By inversion $C \vdash_{\text{rtyp}} e : \exists \alpha{:}\kappa[\gamma_1].\tau_1$, $C_R; \rho{:}\mathtt{R}, \alpha{:}\kappa; \Gamma, x{:}(\tau_1, \rho); \gamma, \epsilon{<}\rho, \gamma_1; \epsilon \cup \rho; \tau \vdash_{\text{styp}} s_1$, $C_R; \cdot \vdash_{\text{k}} \tau : \mathtt{A}$, $R_P \vdash_{\text{epop}} e$,

and $\cdot \vdash_{\text{spop}} s_1$. Only DS4.7 or DS4.11 applies. For DS4.7, the situation is local, $e = \mathsf{pack}\ \tau_2, v\ \mathsf{as}\ \exists\alpha{:}\kappa[\gamma_1].\tau_1$, and s becomes $\mathsf{let}\ \rho, x = v;\ s_1[\tau_2/\alpha]$. By inversion, $C \vdash_{\text{rtyp}} v : \tau_1[\tau_2/\alpha]$, $C_R; \cdot \vdash_{\text{k}} \tau_2 : \kappa$, $\gamma \vdash_{\text{eff}} \gamma_1[\tau_2/\alpha]$, and $R_P \vdash_{\text{epop}} v$. By the Context Weakening Lemma, $C_R; \rho{:}\mathtt{R} \vdash_{\text{k}} \tau_2 : \kappa$, so by the Substitution Lemma, $C_R; \rho{:}\mathtt{R}; (\Gamma, x{:}(\tau_1, \rho))[\tau_2/\alpha]; (\gamma, \epsilon{<}\rho, \gamma_1)[\tau_2/\alpha]; (\epsilon \cup \rho)[\tau_2/\alpha]; \tau[\tau_2/\alpha] \vdash_{\text{styp}} s_1[\tau_2/\alpha]$ and $\cdot \vdash_{\text{spop}} s_1[\tau_2/\alpha]$. The Typing Well-Formedness Lemma ensures $\vdash_{\text{wf}} C$, so the Useless Substitution Lemma ensures
$C_R; \rho{:}\mathtt{R}; \Gamma, x{:}(\tau_1[\tau_2/\alpha], \rho); \gamma, \epsilon{<}\rho, (\gamma_1[\tau_2/\alpha]); \epsilon \cup \rho; \tau \vdash_{\text{styp}} s_1[\tau_2/\alpha]$.
Because $\gamma, \epsilon{<}\rho \vdash_{\text{eff}} \gamma, \epsilon{<}\rho, (\gamma_1[\tau_2/\alpha])$, the Term Weakening Lemma ensures
$C_R; \rho{:}\mathtt{R}; \Gamma, x{:}(\tau_1[\tau_2/\alpha], \rho); \gamma, \epsilon{<}\rho; \epsilon \cup \rho; \tau \vdash_{\text{styp}} s_1[\tau_2/\alpha]$.
So $C; \tau \vdash_{\text{styp}} \mathsf{let}\ \rho, x = v;\ s_1[\tau_2/\alpha]$ and $R_P \vdash_{\text{spop}} \mathsf{let}\ \rho, x = v;\ s_1[\tau_2/\alpha]$.

For DS4.11, the situation is inductive and e becomes e'. By induction $C' \vdash_{\text{rtyp}} e' : \exists\alpha{:}\kappa[\gamma_1].\tau_1$ and $R'_P \vdash_{\text{epop}} e'$. By the Term Weakening Lemma, $C'_R; \rho{:}\mathtt{R}; \Gamma', x{:}(\tau_1[\tau_2/\alpha], \rho); \gamma', \epsilon{<}\rho; \epsilon{\cup}\rho; \tau \vdash_{\text{styp}} s_1[\tau_2/\alpha]$. By the Context Weakening Lemma, $C'_R; \cdot \vdash_{\text{k}} \tau : \mathtt{A}$. So $C'; \tau \vdash_{\text{styp}} \mathsf{open}\ e'\ \mathsf{as}\ \rho, \alpha, x;\ s_1$ and $R'_P \vdash_{\text{spop}} \mathsf{open}\ e'\ \mathsf{as}\ \rho, \alpha, x;\ s_1$.

- SS4.8: Let $s = \mathsf{region}\ \rho, x\ s_1$. Only DS4.8 applies. By inversion
$C_R; \rho{:}\mathtt{R}; \Gamma, x{:}(\text{region}(\rho), \rho); \gamma, \epsilon{<}\rho; \epsilon \cup \rho; \tau \vdash_{\text{styp}} s_1$, $R_P = \cdot$, $\cdot \vdash_{\text{spop}} s_1$, and $\vdash_{\text{hind}} S_G; S_E; \cdot; R_G; R_E; \cdot; \Gamma; \gamma; \epsilon$. Let $S'_G = S_G$, $S'_E = S_E$, $S'_P = i{:}x \mapsto \mathsf{rgn}\ i$, $R'_G = R_G$, $R'_P = i$, and $\Gamma' = \Gamma, x{:}(\text{region}(\text{S}(i)), \text{S}(i))$. Let $\gamma' = \gamma, i_n{<}i$ where $R_E = i_1, \ldots, i_n$ (if $n = 0$, then $\gamma' = \gamma$). By the Heap-Type Weakening Lemma, $C'_R; \Gamma'; \gamma' \vdash_{\text{htyp}} S'_G S'_E \cdot : \Gamma$, so we can derive $C'_R; \Gamma'; \gamma' \vdash_{\text{htyp}} S'_G S'_E S'_P : \Gamma'$ and therefore $\vdash_{\text{hind}} S'_G; S'_E; \cdot; R'_G; R_E; R'_P; \Gamma'; \gamma'; \epsilon$. From $\cdot \vdash_{\text{spop}} s_1$, we can derive $R'_P \vdash_{\text{spop}} s_1; \mathsf{pop}\ i$, so the Substitution Lemma ensures $R'_P \vdash_{\text{spop}} (s_1; \mathsf{pop}\ i)[\text{S}(i)/\rho]$. Our remaining obligation is $C'; \tau \vdash_{\text{styp}} (s_1; \mathsf{pop}\ i)[\text{S}(i)/\rho]$.
The Context Weakening Lemma ensures
$C'_R; \rho{:}\mathtt{R}; \Gamma, x{:}(\text{region}(\rho), \rho); \gamma, \epsilon{<}\rho; \epsilon \cup \rho; \tau \vdash_{\text{styp}} s_1$,
so the Substitution Lemma, Useless Substitution Lemma and $\vdash_{\text{wf}} C$ ensure
$C'_R; \cdot; \Gamma, x{:}(\text{region}(\text{S}(i)), i); \gamma, \epsilon{<}i; \epsilon \cup i; \tau \vdash_{\text{styp}} s_1[\text{S}(i)/\rho]$.
The New-Region Preservation Lemma and Context Weakening Lemma ensure $C'_R; \cdot; \Gamma, x{:}(\text{region}(\text{S}(i)), i); \gamma'; \epsilon \cup i; \tau \vdash_{\text{styp}} s_1[\text{S}(i)/\rho]$, from which we can derive $C'; \tau \vdash_{\text{styp}} (s_1; \mathsf{pop}\ i)[\text{S}(i)/\rho]$.

- SS4.9: Let $s = s_1; \mathsf{pop}\ i$. By inversion $C_R; \cdot; \Gamma; \gamma; \epsilon \cup i; \tau \vdash_{\text{styp}} s_1$ and $R_1 \vdash_{\text{spop}} s_1$ where $R_P = i, R_1$. Only DS4.9, DS4.10, and DS4.12 apply. For DS4.9 and DS4.10, $s' = s_1$. Inversion of $R_1 \vdash_{\text{spop}} s_1$ and the Values Effectless Lemma ensure $R_1 = \cdot$, so $S_P = i{:}H$ for some H. Letting $S'_G = S_G, i{:}H$, $S'_E = S_E$, $S'_P = \cdot$, $R'_G = R_G, i$, $R'_P = \cdot$, and $\Gamma' = \Gamma$, conclusions 1–3 hold. Inverting the $\vdash_{\text{hind}}$ assumption, γ has the form $\gamma_G\gamma_1$ where γ_1 has the form $\gamma'_1, i_{n-1}{<}i_n$.

Letting $\gamma'_G = \gamma_G, i_{n-1}{<}i_n$ and $\gamma' = \gamma'_1\gamma'_G$, a simple induction on n shows that conclusion 4 holds. A simple induction on the assumed $\vdash_{\text{htyp}}$ derivation ensures $C_R; \Gamma; \gamma \vdash_{\text{htyp}} S'_G S'_E S'_P : \Gamma$ (intuitively, heap typing is reorderable). Because $R'_G R_E R'_P \subseteq R_G R_E R_P$ (in fact, the sets are equal), the Typing Weakening Lemma ensures $C'_R; \Gamma; \gamma' \vdash_{\text{htyp}} S'_G S'_E S'_P : \Gamma$. So by choosing i_{n-1} for the ϵ that i outlives, we can conclude $\vdash_{\text{hind}} S'_G; S'_E; S'_P; R'_G; R_E; R'_P; \gamma'; \epsilon$. By the Values Effectless Lemma and inversion of the typing derivation for s_1, $C_R; \cdot; \Gamma; \gamma; \epsilon; \tau \vdash_{\text{styp}} s_1$. So the Typing Weakening Lemma ensures $C'; \tau \vdash_{\text{styp}} s_1$. Because $R'_P \vdash_{\text{spop}} s_1$, we can conclude $\vdash_{\text{sind}} S'_G; S'_E; S'_P; \tau; s_1 : R'_G; R_E; R'_P; \Gamma; \gamma'$.

For DS4.12, s_1 becomes s'_1. To apply the induction hypothesis, we need an appropriate $\vdash_{\text{hind}}$ fact to use with $C_R; \cdot; \Gamma; \gamma; \epsilon \cup i; \tau \vdash_{\text{styp}} s_1$ and $R_1 \vdash_{\text{spop}} s_1$. Inverting the $\vdash_{\text{hind}}$ assumption ensures S_P has the form $i{:}H, S_1$. Letting $S_{E1} = S_E, i{:}H$ and $R_{E1} = R_E, i$, we observe $R_G R_{E1} R_1 = R_G R_E R_P$ and $S_G S_{E1} S_1 = S_G S_E S_P$, so we can derive $\vdash_{\text{hind}} S_G; S_{E1}; S_1; R_G; R_{E1}; R_1; \Gamma; \gamma; \epsilon \cup i$. The induction hypothesis and inversion ensure $\vdash_{\text{hind}} S'_G; S'_{E1}; S'_1; R'_G; R_{E1}; R'_1; \Gamma'; \gamma'; \epsilon \cup i$, $R'_G R_{E1} R'_1; \cdot; \Gamma'; \gamma'; \epsilon \cup i; \tau \vdash_{\text{styp}} s'_1$, and $R'_1 \vdash_{\text{spop}} s'_1$ with conclusions 1–4 holding. Inverting the $\vdash_{\text{hind}}$ fact from the induction ensures S'_{E1} has the form $S'_E, i{:}H'$. Letting $S'_P = i{:}H', S'_1$ and $R'_P = i, R'_P$, we observe $R'_G R_E R'_P = R'_G R_{E1} R'_1$ and $S'_G S'_E S'_P = S'_G S'_{E1} S'_1$, so we can derive $\vdash_{\text{hind}} S_G; S'_E; S'_P; R'_G; R_E; R'_P; \Gamma'; \gamma'; \epsilon$. From $R'_G R_{E1} R'_1; \cdot; \Gamma'; \gamma'; \epsilon \cup i; \tau \vdash_{\text{styp}} s'_1$ we can derive $C'; \tau \vdash_{\text{styp}} s'_1; \mathsf{pop}\ i$. From $R'_1 \vdash_{\text{spop}} s'_1$ we can derive $R'_P \vdash_{\text{spop}} s'_1; \mathsf{pop}\ i$. So conclusion 5 holds. It is easy to verify the other conclusions.

Lemma B.19 (Type and Pop Progress).

1. *If $\vdash_{\text{sind}} S_G; S_E; S_P; \tau; s : R_G; R_E; R_P; \Gamma$, then $s = v$ for some v, $s = \mathsf{return}\ v$ for some v, or there exists S'_G, S', and s' such that $S_G; S_E S_P; s \xrightarrow{\text{s}} S'_G; S'; s'$.*

2. *If $\vdash_{\text{rind}} S_G; S_E; S_P; e : \tau; R_G; R_E; R_P; \Gamma$, then $e = v$ for some v or there exists S'_G, S', and e' such that $S_G; S_E S_P; e \xrightarrow{\text{r}} S'_G; S'; e'$.*

3. *If $\vdash_{\text{lind}} S_G; S_E; S_P; e : \tau, r; R_G; R_E; R_P; \Gamma$, then $e = xp$ for some x and p or there exists S'_G, S', and e' such that $S_G; S_E S_P; e \xrightarrow{\text{l}} S'_G; S'; e'$.*

Proof: The proofs are by simultaneous induction on the typing derivations implied by the $\vdash_{\text{sind}}$, $\vdash_{\text{rind}}$, and $\vdash_{\text{lind}}$ assumptions, proceeding by cases on the last rule used. In each case, let $C = R_G R_E R_P; \cdot; \Gamma; \gamma; \epsilon$.

- SL4.1: This case is trivial because e has the form xp.

- SL4.2: Let $e = *e_1$. If e_1 is a value, the Canonical Forms Lemma ensures it has the form $\&xp$, so DL4.2 applies. Else inversion ensures $C \vdash_{\text{rtyp}} e_1 : \tau{*}r$ and $R_P \vdash_{\text{epop}} e_1$, so the result follows from induction and DL4.3.

- SL4.3–4: Let $e = e_1.i$. If e_1 has the form xp, then DL4.1 applies. Else inversion ensures $C \vdash_{\text{ltyp}} e_1 : \tau_0 \times \tau_1, r$ and $R_P \vdash_{\text{epop}} e_1$, so the result follows from induction and DL4.4.

- SL4.5: This case follows from induction.

- SR4.1: Let $e = xp$. The Access Control Lemma ensures $x \in \text{Dom}(H)$ for some H in S_E. The $\vdash_{\text{hind}}$ hypotheses, the Heap-Type Well-Formedness Lemma, and the Values Effectless Lemma ensure $C \vdash_{\text{rtyp}} H(x) : \tau'$ (where $\Gamma(x) = (\tau', r)$). So Heap-Object Safety Lemma 4 ensures DR4.1 applies.

- SR4.2: This case is analogous to case SL4.2, using DR4.3 for DL4.2 and DR4.11 for DL4.3.

- SR4.3–4: Let $e = e_1.i$. If e_1 is a value, the Canonical Forms Lemma ensures it has the form (v_0, v_1), so DR4.4 applies. Else inversion ensures $C \vdash_{\text{rtyp}} e_1 : \tau_0 \times \tau_1$ and $R_P \vdash_{\text{epop}} e_1$, so the result follow from induction and DR4.11.

- SR4.5: This case is trivial because e is a value.

- SR4.6: Let $e = \&e_1$. If e_1 has the form xp, then e is a value. Else inversion ensures $C \vdash_{\text{ltyp}} e_1 : \tau, r$ and $R_P \vdash_{\text{epop}} e_1$, so the result follows from induction and DR4.10.

- SR4.7: Let $e = (e_0, e_1)$. If e_0 and e_1 are values, then e is a value. Else if e_0 is not a value, inversion ensures $C \vdash_{\text{rtyp}} e_0 : \tau_0$ and $R_P \vdash_{\text{epop}} e_0$, so the result follows from induction and DR4.11. Else inversion ensures $C \vdash_{\text{rtyp}} e_1 : \tau_1$ and $R_P \vdash_{\text{epop}} e_1$, so the result follows from induction and DR4.11.

- SR4.8: Let $e = (e_1{=}e_2)$. If e_1 has the form xp and e_2 is a value, then inversion of the typing derivation ensures $\Gamma(x) = (\tau, r)$, $\gamma; \epsilon \vdash_{\text{acc}} r$, and $\vdash \text{gettype}(\tau, p, \tau')$. So the Access Control Lemma ensures $x \in \text{Dom}(H)$ for some H in S_E. The $\vdash_{\text{hind}}$ hypotheses, the Heap-Type Well-Formedness Lemma, and the Values Effectless Lemma ensure $C \vdash_{\text{rtyp}} H(x) : \tau$. So Heap-Object Safety Lemma 5 ensures DL4.2 applies. Else if e_1 does not have the form xp, inversion ensures $C \vdash_{\text{ltyp}} e_1 : \tau, r$ and $R_P \vdash_{\text{epop}} e_1$, so the result follows from induction and DR4.10. Else inversion ensures $C \vdash_{\text{rtyp}} e_2 : \tau$ and $R_P \vdash_{\text{epop}} e_2$, so the result follows from induction and DR4.11.

- SR4.9: Let $e = e_1(e_2)$. If e_1 and e_2 are values, the Canonical Forms Lemma ensures e_1 is a function, so DR4.5 applies. Else if e_1 is not a value, inversion ensures $C \vdash_{\text{rtyp}} e_1 : \tau' \xrightarrow{\epsilon'} \tau$ and $R_P \vdash_{\text{epop}} e_1$, so the result follows from induction

and DR4.11. Else inversion ensures $C \vdash_{\text{rtyp}} e_2 : \tau'$ and $R_P \vdash_{\text{epop}} e_2$, so the result follows from induction and DR4.11.

- SR4.10: Let $e = \mathsf{call}\ s$. If $s = \mathsf{return}\ v$, then DR4.6 applies. Else inversion ensures $C; \tau \vdash_{\text{styp}} s$ and $R_P \vdash_{\text{spop}} s$. Because $\vdash_{\text{ret}} s$ ensures s does not have the form v, the result follows from induction and DR4.9.

- SR4.11: Let $e = e_1[\tau']$. If e_1 is a value, the Canonical Forms Lemma ensures it is a polymorphic term, so DR4.7 applies. Else inversion ensures $C \vdash_{\text{rtyp}} e_1 : \tau''$ and $R_P \vdash_{\text{epop}} e_1$, so the result follows from induction and DR4.11.

- SR4.12: Let $e = \mathsf{pack}\ \tau', e_1\ \mathsf{as}\ \tau$. If e_1 is a value, then e is a value. Else inversion ensures $C \vdash_{\text{rtyp}} e_1 : \tau''$ and $R_P \vdash_{\text{epop}} e_1$, so the result follows from induction and DR4.11

- SR4.13–14: These cases are trivial because e is a value.

- SR4.15: Let $e = \mathsf{rnew}\ e_1\ e_2$. If e_1 and e_2 are values, the Canonical Forms Lemma ensures e_1 has the form $\mathsf{rgn}\ i$, so $r = \mathrm{S}(i)$. Because $\gamma; \epsilon \vdash_{\text{acc}} r$, the Access Control Lemma ensures i names a heap in S_E, so α-conversion ensures DR4.8 applies. Else if e_1 is not a value, then inversion ensures $C \vdash_{\text{rtyp}} e_1 : \mathrm{region}(r)$ and $R_P \vdash_{\text{epop}} e_1$, so the result follows from induction and DR4.11. Else if e_1 is a value and e_2 is not a value, then inversion ensures $C \vdash_{\text{rtyp}} e_2 : \tau'$ and $R_P \vdash_{\text{epop}} e_2$, so the result follows from induction and DR4.11.

- SR4.16: This case is trivial because e is a value.

- SR4.17: This case follows from induction.

- SS4.1–2: If e is a value, the result is immediate. Else inversion ensures $C \vdash_{\text{rtyp}} e : \tau'$ (for SS4.2 $\tau' = \tau$) and $R_P \vdash_{\text{epop}} e$, so the result follows from induction and DS4.11.

- SS4.3: Let $s = s_1; s_2$. If s_1 is some v, then DS4.2 applies. Else if $s_1 = \mathsf{return}\ v$ for some v, then DS4.3 applies. Else inversion ensures $C; \tau \vdash_{\text{styp}} s_1$ and $R_P \vdash_{\text{spop}} s_1$, so the result follows from induction and DS4.12.

- SS4.4: DS4.6 applies.

- SS4.5: If e is a value, the Canonical Forms Lemma ensures $e = i$ for some i, so either DS4.4 or DS4.5 applies. Else inversion ensures $C \vdash_{\text{rtyp}} e : \mathrm{int}$ and $R_P \vdash_{\text{epop}} e$, so the result follows from induction and DS4.11.

- SS4.6: If e is a value, then α-conversion ensures DS4.1 applies. Else inversion ensures $C \vdash_{\text{rtyp}} e : \tau'$ and $R_P \vdash_{\text{epop}} e$, so the result follows from induction and DS4.11.

- SS4.7: If e is a value, then the Canonical Forms Lemma ensures it is an existential package, so DS4.7 applies. Else inversion ensures $C \vdash_{\text{rtyp}} e : \tau'$ and $R_P \vdash_{\text{epop}} e$, so the result follows from induction and DS4.11.

- SS4.8: Because of α-conversion, DS4.8 applies.

- SS4.9: Let $s = s_1; \mathsf{pop}\ i$. If s_1 is some v, then $R_P \vdash_{\text{spop}} v; \mathsf{pop}\ i$ and the Values Effectless Lemma ensures $R_P = i$. So the $\vdash_{\text{hind}}$ assumptions ensure $S_P = i{:}H$ for some H, i.e., i is the youngest live region. So DS4.9 applies.

  Else if $s_1 = \mathsf{return}\ v$ for some v, then by the same argument as above, DS4.10 applies.

  Else $R_P \vdash_{\text{spop}} s_1$ ensures $R_P = i, R'_P$ for some R'_P and $\underline{R'_P \vdash_{\text{spop}} s_1}$. Given the assumptions of the $\vdash_{\text{hind}}$ derivation, $i = i_{j+1}$ and $R'_P = \underline{i_{j+2}, \ldots, i_n}$. Letting $S'_E = S_E, i_{j+1}{:}H_{j+1}$, $S'_P = i_{j+2}{:}H_{j+2}, \ldots, i_n{:}H_n$, and $R'_E = i_1, \ldots, i_{j+1}$, we can use the $\vdash_{\text{hind}}$ assumptions to derive $\underline{\vdash_{\text{hind}} S_G; S'_E; S'_P; R_G; R'_E; R'_P; \Gamma; \gamma\gamma_G; \epsilon \cup i_{j+1}}$. (In particular, $R_G R'_E R'_P = R_G \underline{R_E R_P}$ and $S_G S'_E S'_P = \underline{S_G S_E S_P}$.) Inverting the original typing derivation ensures $\underline{R_G R'_E R'_P; \cdot; \Gamma; \gamma\gamma_G; \epsilon \cup i_{j+1} \vdash_{\text{rtyp}} \tau : s_1}$. Given the underlined conclusions, induction ensures s_1 can take a step, so the result follows from DS4.12.

Appendix C

Chapter 5 Safety Proof

This appendix proves Theorem 5.2, which we repeat here:

Definition 5.1. *A program $P = (H; L; L_0; T_1 \cdots T_n)$ is badly stuck if it has a badly stuck thread. A badly stuck thread is a thread (L', s) in P such that there is no v such that $s = \mathsf{return}\ v$ and $L = \cdot$; and there is no i such that $H; (L; L_0, i; L'); s \xrightarrow{s} H'; \overline{L}'; s_{\text{opt}}; s'$ for some H', $\overline{L}'$, s_{opt}, and s'.*

Theorem 5.2 (Type Safety). *If $\cdot; \cdot; \cdot; \cdot; \emptyset; \tau \vdash_{\text{styp}} s$, $\vdash_{\text{ret}} s$, s is junk-free, s has no release statements, and $\cdot; (\cdot; \cdot; \cdot); (\cdot; s) \rightarrow^* P$ (where $\rightarrow^*$ is the reflexive transitive closure of $\rightarrow$), then P is not badly stuck.*

Before presenting and proving the necessary lemmas in "bottom-up" order, we summarize the structure of the argument. It is similar to the proof in Chapter 4, but it is simpler because locks are unordered and more complicated because of junk expressions.

The Type Soundness Theorem is a simple corollary given the Preservation and Progress Lemmas. In turn, these lemmas follow from the Type and Release Preservation (and Return Preservation) and Type and Release Progress Lemmas, respectively. These lemmas establish type preservation and progress for an individual thread by strengthening their claims to apply inductively to every well-typed statement and expression (given an appropriate type context). Given the intricacy of $\vdash_{\text{prog}} P$, the necessary assumptions for a statement or expression are complicated enough that we define the judgments in Figure C.1 to describe them accurately and concisely.

These rules merge the locks held by other threads and the shared heap locations guarded by such locks into one H_X and L_X. Furthermore, it does not suffice to say $L_i \vdash_{\text{srel}} s$ (or $L_i \vdash_{\text{erel}} e$) where L_i describes the locks held by the thread containing s. Instead, we must distinguish the locks released by statements containing s

(call these L_E) and locks in release statements contained in s (call these L_R). We require $L_i = L_E L_R$ and $L_R \vdash_{\text{srel}} s$. We use L_E to determine the ϵ used to type-check s. (For "top-level" statements, $\epsilon = \emptyset$.) It is really these judgments that capture exactly what a statement or expression reduction preserves. The interesting part of each case of the preservation proof is which of the arguments to the judgment (for example, L_R or H_{0S}) change in order to prove the result of the reduction satisfies the property.

The Return Preservation Lemma is used in the proof of the Preservation Lemma to show that threads always return. It is also used in case DR5.9 of the Type and Release Preservation Lemma.

The Access Control Lemma establishes that if the static context permits a term to access a heap location, then that location is local to the thread or guarded by a lock that the thread holds. We use this lemma in cases DR5.1 and DR5.2A of the Type and Release Preservation Lemma to argue that the heap accessible to the executing thread is junk-free.

The Sharable Values Need Only Sharable Context Lemma establishes that if a value has some type of kind `AS` in some context, then the value has the same type in a context where all unsharable locations are omitted. Intuitively, if we needed any of these locations to type-check the value, then the value is not sharable. We use this lemma in case DS5.12 of the Type and Release Preservation Lemma because spawning a thread involves "moving" two values to a different thread. We also use this lemma in cases DR5.2A and DS5.1 when the assigned-to location is sharable because assignment involves "moving" a value into the heap (in this case, part of the heap that must type-check without using unsharable locations).

The Canonical Forms Lemma describes the form of "top-level" values (where type variables are never in scope). As usual, we use this lemma throughout the Type and Release Progress Lemma proof to argue about the form of values given their types.

The Term Substitution Lemmas establish that proper substitution of types through terms preserves important properties. As expected, we use these lemmas for cases involving substitution (DR5.7 and DS5.7) in the preservation proofs.

The Heap-Type Well-Formedness Lemma provides some rather obvious properties of all locations in a well-typed heap. We use these properties in subsequent proofs when we need to conclude properties about a location x knowing only that x is in a heap that type-checks under some context.

The Values Effectless Lemma provides properties about values. We use these properties in subsequent proofs to refine the information provided by assumptions. For example, when proving type preservation for rule DR5.4, we use the lemma to conclude $L_R = \cdot$, so we can derive $L_R \vdash_{\text{erel}} v_0$ and $L_R \vdash_{\text{erel}} v_1$.

The Typing Well-Formedness Lemma shows that the typing rules have enough

well-formedness hypotheses to conclude that the context and the result types are always well-formed and have the right kinds. We use this lemma to conclude context well-formedness when we need it as an explicit assumption to type-check the result of an evaluation step or to apply various weakening lemmas. We also use this lemma to conclude the kinds of types in typing judgments (which is sometimes necessary to establish the assumptions of other lemmas, such as the Type Substitution Lemmas).

The Type Substitution Lemmas show how various type-level properties are preserved under appropriate type substitutions. These lemmas are necessary to prove the Term Substitution Lemmas and case SR5.11 of the Typing Well-Formedness Lemma.

The Commuting Substitutions Lemma is necessary as usual for polymorphic languages with type-substitution, as previous chapters have demonstrated. As in Chapter 4, the proof is slightly nontrivial because of the definition of substitution through effects.

The Type Canonical Forms Lemma restricts the form of types with kind `LS`. We use the lemma to restrict the form of ℓ when proving properties about $\gamma; \epsilon \vdash_{\text{acc}} \ell$ and in case DS5.1 of the Type and Release Preservation Lemma proof. It also provides results needed to prove the Typing Well-Formedness Lemma and the Term Substitution Lemma. These results would be immediate if we did not have subkinding.

The Useless Substitution Lemmas are all obvious. We use them to show properties are preserved under substitution when we know that part of the static context does not contain the substituted-for type variable. Specifically, case SR5.13 of Term Substitution Lemma 4 needs this lemma because the function's free variables are heap locations (which must have closed types). Similarly, the cases of the Type and Release Preservation Lemma proof that use substitution use the Useless Substitution Lemma to obtain an appropriate context for type-checking the result of the evaluation step.

Finally, the various weakening lemmas serve their usual purpose in the preservation proofs. Reduction steps can extend the heap or the set of allocated locks, which provides a larger context for type-checking other values (and in the case of locks, for kind-checking types, etc.). Weakening ensures that enlarging a context cannot make a value fail to type-check. The structure of our preservation argument produces additional needs for weakening. For example, when reading a sharable value from the heap, the value is "copied" from a place where it type-checked with reference only to sharable values to a term that can also refer to thread-local values. We also use weakening to type-check terms under a context with more permissive ϵ and γ; typically, explicit assumptions provide that ϵ and γ are more permissive and we cannot use the less permissive ones because of other terms that

must still type-check.

We omit some uninteresting proofs, most of which are analogous to corresponding proofs in Chapter 4. We still state as lemmas all facts that require inductive arguments.

Lemma C.1 (Context Weakening).

1. *If* $L; \Delta \vdash_{\text{wf}} \epsilon$, *then* $LL'; \Delta\Delta' \vdash_{\text{wf}} \epsilon$.
2. *If* $L; \Delta \vdash_{\text{wf}} \gamma$, *then* $LL'; \Delta\Delta' \vdash_{\text{wf}} \gamma$.
3. *If* $L; \Delta \vdash_{\text{k}} \tau : \kappa$, *then* $LL'; \Delta\Delta' \vdash_{\text{k}} \tau : \kappa$.
4. *If* $L; \Delta \vdash_{\text{wf}} \Gamma$, *then* $LL'; \Delta\Delta' \vdash_{\text{wf}} \Gamma$.
5. *If* $\gamma \vdash_{\text{eff}} \epsilon_1 \subseteq \epsilon_2$ *and* $\gamma' \vdash_{\text{eff}} \gamma$ *then* $\gamma' \vdash_{\text{eff}} \epsilon_1 \subseteq \epsilon_2$.
6. *If* $\gamma; \epsilon \vdash_{\text{acc}} \ell$, $\gamma' \vdash_{\text{eff}} \gamma$, *and* $\gamma' \vdash_{\text{eff}} \epsilon \subseteq \epsilon'$, *then* $\gamma'; \epsilon' \vdash_{\text{acc}} \ell$.
7. *If* $\gamma_2 \vdash_{\text{eff}} \gamma_1$, *and* $\gamma_3 \vdash_{\text{eff}} \gamma_2$ *then* $\gamma_3 \vdash_{\text{eff}} \gamma_1$.
8. *If* $L \vdash_{\text{shr}} \Gamma$, *then* $LL' \vdash_{\text{shr}} \Gamma$.
9. *If* $L \vdash_{\text{loc}} \Gamma$ *and* $L; \cdot \vdash_{\text{wf}} \Gamma$, *then* $LL' \vdash_{\text{loc}} \Gamma$.

Lemma C.2 (Term Weakening). *Suppose* $\vdash_{\text{wf}} LL'; \Delta\Delta'; \Gamma\Gamma'; \gamma'; \epsilon'$, $\gamma' \vdash_{\text{eff}} \gamma$, *and* $\gamma' \vdash_{\text{eff}} \epsilon \subseteq \epsilon'$.

1. *If* $L; \Delta; \Gamma; \gamma; \epsilon \vdash_{\text{ltyp}} e : \tau, \ell$, *then* $LL'; \Delta\Delta'; \Gamma\Gamma'; \gamma'; \epsilon' \vdash_{\text{ltyp}} e : \tau, \ell$.
2. *If* $L; \Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} e : \tau$, *then* $LL'; \Delta\Delta'; \Gamma\Gamma'; \gamma'; \epsilon' \vdash_{\text{rtyp}} e : \tau$.
3. *If* $L; \Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{\text{styp}} s$, *then* $LL'; \Delta\Delta'; \Gamma\Gamma'; \gamma'; \epsilon'; \tau \vdash_{\text{styp}} s$.

Lemma C.3 (Heap-Type Weakening). *Suppose* $LL'; \cdot \vdash_{\text{wf}} \Gamma\Gamma'$.

1. *If* $L; \Gamma \vdash_{\text{htyp}} H : \Gamma''$, *then* $LL'; \Gamma\Gamma' \vdash_{\text{htyp}} H : \Gamma''$.
2. *If* $\Gamma; L \vdash_{\text{hlk}} H$, *then* $\Gamma\Gamma'; LL' \vdash_{\text{hlk}} H$.

Lemma C.4 (Useless Substitution). *Suppose* $\alpha \notin \text{Dom}(\Delta)$.

1. *If* $L; \Delta \vdash_{\text{wf}} \epsilon$, *then* $\epsilon[\tau/\alpha] = \epsilon$.
2. *If* $L; \Delta \vdash_{\text{wf}} \gamma$, *then* $\gamma[\tau/\alpha] = \gamma$.
3. *If* $L; \Delta \vdash_{\text{k}} \tau' : \kappa$, *then* $\tau'[\tau/\alpha] = \tau'$.

4. *If* $L;\Delta \vdash_{\text{wf}} \Gamma$, *then* $\Gamma[\tau/\alpha] = \Gamma$.

Lemma C.5 (Type Canonical Forms).

1. *If* $L;\Delta \vdash_{\text{k}} \tau : \mathtt{L}\sigma$, *then* $\tau = \mathrm{S}(i)$ *for some* $i \in L$, *or* $\tau = loc$, *or* $\tau = \alpha$ *for some* $\alpha \in \mathrm{Dom}(\Delta)$.

2. *If* $L;\Delta \vdash_{\text{k}} \tau * \ell : \kappa$, *then* $L;\Delta \vdash_{\text{k}} \tau : \mathtt{AU}$ *and* $L;\Delta \vdash_{\text{k}} \tau : \mathtt{LU}$. *Furthermore, if* $\kappa = \theta\mathtt{S}$, *then* $L;\Delta \vdash_{\text{k}} \tau : \mathtt{AS}$ *and* $L;\Delta \vdash_{\text{k}} \tau : \mathtt{LS}$.

3. *If* $L;\Delta \vdash_{\text{k}} \tau_0 \times \tau_1 : \kappa$, *then* $L;\Delta \vdash_{\text{k}} \tau_0 : \mathtt{AU}$ *and* $L;\Delta \vdash_{\text{k}} \tau_1 : \mathtt{AU}$.

4. *If* $L;\Delta \vdash_{\text{k}} \tau' \xrightarrow{\epsilon} \tau : \kappa$, *then* $L;\Delta \vdash_{\text{k}} \tau : \mathtt{AU}$ *and* $L;\Delta \vdash_{\text{k}} \tau' \xrightarrow{\epsilon} \tau : \mathtt{AS}$.

5. *If* $L;\Delta \vdash_{\text{k}} \forall\alpha{:}\kappa[\gamma].\tau : \kappa'$, *then* $L;\Delta,\alpha{:}\kappa \vdash_{\text{k}} \tau : \mathtt{AU}$.

6. *If* $L;\Delta \vdash_{\text{k}} \mathrm{lock}(\ell) : \kappa$, *then* $L;\Delta \vdash_{\text{k}} \ell : \mathtt{LU}$.

Proof: Each proof is by induction on the assumed kinding derivation. Induction is necessary only because the last step in the derivation may be subsumption. For the noninductive cases (except for the first lemma), we use subsumption to derive that the type(s) in the conclusion have kind AU or LU.

Lemma C.6 (Commuting Substitutions). *Suppose* β *is not free in* τ_2.

1. $\epsilon[\tau_1/\beta][\tau_2/\alpha] = \epsilon[\tau_2/\alpha][\tau_1[\tau_2/\alpha]/\beta]$.

2. $\gamma[\tau_1/\beta][\tau_2/\alpha] = \gamma[\tau_2/\alpha][\tau_1[\tau_2/\alpha]/\beta]$.

3. $\tau_0[\tau_1/\beta][\tau_2/\alpha] = \tau_0[\tau_2/\alpha][\tau_1[\tau_2/\alpha]/\beta]$.

Lemma C.7 (Type Substitution). *Suppose* $L;\Delta \vdash_{\text{k}} \tau : \kappa$.

1. $L;\Delta \vdash_{\text{wf}} \mathrm{locks}(\tau)$

2. *If* $L;\Delta,\alpha{:}\kappa \vdash_{\text{wf}} \epsilon$, *then* $L;\Delta \vdash_{\text{wf}} \epsilon[\tau/\alpha]$.

3. *If* $L;\Delta,\alpha{:}\kappa \vdash_{\text{wf}} \gamma$, *then* $L;\Delta \vdash_{\text{wf}} \gamma[\tau/\alpha]$.

4. *If* $L;\Delta,\alpha{:}\kappa \vdash_{\text{k}} \tau' : \kappa'$, *then* $L;\Delta \vdash_{\text{k}} \tau'[\tau/\alpha] : \kappa'$.

5. *If* $L;\Delta,\alpha{:}\kappa \vdash_{\text{wf}} \Gamma$, *then* $L;\Delta \vdash_{\text{wf}} \Gamma[\tau/\alpha]$.

6. *If* $\vdash_{\text{wf}} L;\Delta,\alpha{:}\kappa;\Gamma;\gamma;\epsilon$, *then* $\vdash_{\text{wf}} L;\Delta;\Gamma[\tau/\alpha];\gamma[\tau/\alpha];\epsilon[\tau/\alpha]$.

7. *If* $\gamma \vdash_{\text{eff}} \epsilon_1 \subseteq \epsilon_2$, *then* $\gamma[\tau/\alpha] \vdash_{\text{eff}} \epsilon_1[\tau/\alpha] \subseteq \epsilon_2[\tau/\alpha]$.

8. *If* $\gamma; \epsilon \vdash_{\text{acc}} \ell$ *and* $L; \Delta, \alpha{:}\kappa \vdash_{\text{k}} \ell : \texttt{LU}$, *then* $\gamma[\tau/\alpha]; \epsilon[\tau/\alpha] \vdash_{\text{acc}} \ell[\tau/\alpha]$.

9. *If* $\gamma \vdash_{\text{eff}} \gamma'$, *then* $\gamma[\tau/\alpha] \vdash_{\text{eff}} \gamma'[\tau/\alpha]$.

Lemma C.8 (Typing Well-Formedness).

1. *If* $C \vdash_{\text{ltyp}} e : \tau, \ell$, *then* $\vdash_{\text{wf}} C$, $C_L; C_\Delta \vdash_{\text{k}} \tau : \texttt{AU}$, *and* $C_L; C_\Delta \vdash_{\text{k}} \ell : \texttt{LU}$.

2. *If* $C \vdash_{\text{rtyp}} e : \tau$, *then* $\vdash_{\text{wf}} C$ *and* $C_L; C_\Delta \vdash_{\text{k}} \tau : \texttt{AU}$.

3. *If* $C; \tau \vdash_{\text{styp}} s$, *then* $\vdash_{\text{wf}} C$. *If* $C; \tau \vdash_{\text{styp}} s$ *and* $\vdash_{\text{ret}} s$, *then* $C_L; C_\Delta \vdash_{\text{k}} \tau : \texttt{AU}$.

Proof: We omit most of the proof. It is by simultaneous induction on the assumed typing derivations. Cases where the result type is part of a hypotheses' result type (SL5.2, SR5.2, SR5.3, SR5.4, SR5.9, SR5.11) use the Type Canonical Forms Lemma. In Chapter 4, the analogous results were established directly in the Typing Well-Formedness Lemma proof because that chapter had less subkinding.

Lemma C.9 (Heap-Type Well-Formedness). *If* $L; \Gamma' \vdash_{\text{htyp}} H : \Gamma$, *then* $L; \cdot \vdash_{\text{wf}} \Gamma$ *and* $\text{Dom}(\Gamma) = \text{Dom}(H)$. *Furthermore, for all* $x \in \text{Dom}(H)$, $L; \cdot; \Gamma'; \cdot; \emptyset \vdash_{\text{rtyp}} H(x) : \tau$ *where* $\Gamma(x) = (\tau, \ell)$ *for some* ℓ *and* $\cdot \vdash_{\text{erel}} H(x)$.

Lemma C.10 (Term Substitution).

1. *If* $\vdash_{\text{ret}} s$, *then* $\vdash_{\text{ret}} s[\tau/\alpha]$.

2. *If* $L \vdash_{\text{srel}} s$, *then* $L \vdash_{\text{srel}} s[\tau/\alpha]$.
   If $L \vdash_{\text{erel}} e$, *then* $L \vdash_{\text{erel}} e[\tau/\alpha]$.

3. *If* $\vdash_{\text{jf}} s$, *then* $\vdash_{\text{jf}} s[\tau/\alpha]$.
   If $\vdash_{\text{jf}} e$, *then* $\vdash_{\text{jf}} e[\tau/\alpha]$.

4. *Suppose* $L; \Delta \vdash_{\text{k}} \tau : \kappa$.
   If $L; \Delta, \alpha{:}\kappa; \Gamma; \gamma; \epsilon \vdash_{\text{ltyp}} e : \tau', \ell$,
   then $L; \Delta; \Gamma[\tau/\alpha]; \gamma[\tau/\alpha]; \epsilon[\tau/\alpha] \vdash_{\text{ltyp}} e[\tau/\alpha] : \tau'[\tau/\alpha], \ell[\tau/\alpha]$.
   If $L; \Delta, \alpha{:}\kappa; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} e : \tau'$,
   then $L; \Delta; \Gamma[\tau/\alpha]; \gamma[\tau/\alpha]; \epsilon[\tau/\alpha] \vdash_{\text{rtyp}} e[\tau/\alpha] : \tau'[\tau/\alpha]$.
   If $L; \Delta, \alpha{:}\kappa; \Gamma; \gamma; \epsilon; \tau' \vdash_{\text{styp}} s$,
   then $L; \Delta; \Gamma[\tau/\alpha]; \gamma[\tau/\alpha]; \epsilon[\tau/\alpha]; \tau'[\tau/\alpha] \vdash_{\text{styp}} s[\tau/\alpha]$.

Proof: We omit the proofs because they are either analogous to proofs in Chapter 4 or trivial inductive arguments. However, we mention two unusual cases in proving the last lemma (by simultaneous induction on the assumed typing derivations). In case SR5.13, the assumption $L; \cdot \vdash_{\text{wf}} \Gamma_1$ and the Useless Substitution Lemma ensure $\Gamma_1[\tau/\alpha] = \Gamma_1$, so we can use $L; \cdot \vdash_{\text{wf}} \Gamma_1$ and $L \vdash_{\text{shr}} \Gamma_1$ to derive

$$\frac{\begin{array}{l}\vdash_{\text{jf}} H_{0S} \qquad \Gamma_S; L_0 \vdash_{\text{hlk}} H_{0S} \qquad \Gamma_S; L_X \vdash_{\text{hlk}} H_{XS} \qquad \Gamma_S; L_R L_E \vdash_{\text{hlk}} H_S \\ L; \Gamma_S \vdash_{\text{htyp}} H_{XS} H_{0S} H_S : \Gamma_S \quad L \vdash_{\text{shr}} \Gamma_S \\ L; \Gamma_S \Gamma_U \vdash_{\text{htyp}} H_U : \Gamma_U \qquad L \vdash_{\text{loc}} \Gamma_U \\ L = L_0 L_X L_R L_E \qquad L_E = i_1, \ldots, i_n \qquad \epsilon = i_1 \cup \ldots \cup i_n\end{array}}{\vdash_{\text{hind}} H_{XS}; H_{0S}; H_S; H_U; L; L_0; L_X; L_R; L_E : \Gamma_S; \Gamma_U; \epsilon}$$

$$\frac{\begin{array}{l}\vdash_{\text{hind}} H_{XS}; H_{0S}; H_S; H_U; L; L_0; L_X; L_R; L_E : \Gamma_S; \Gamma_U; \epsilon \\ \vdash_{\text{j}} H_S H_U; s \qquad L; \cdot; \Gamma_S \Gamma_U; \cdot; \epsilon; \tau \vdash_{\text{styp}} s \qquad L_R \vdash_{\text{srel}} s\end{array}}{\vdash_{\text{sind}} H_{XS}; H_{0S}; H_S; H_U; L; L_0; L_X; L_R; L_E; \tau; s : \Gamma_S; \Gamma_U}$$

$$\frac{\begin{array}{l}\vdash_{\text{hind}} H_{XS}; H_{0S}; H_S; H_U; L; L_0; L_X; L_R; L_E : \Gamma_S; \Gamma_U; \epsilon \\ \vdash_{\text{j}} H_S H_U; e \qquad L; \cdot; \Gamma_S \Gamma_U; \cdot; \epsilon \vdash_{\text{rtyp}} e : \tau \qquad L_R \vdash_{\text{erel}} e\end{array}}{\vdash_{\text{rind}} H_{XS}; H_{0S}; H_S; H_U; L; L_0; L_X; L_R; L_E; e : \tau; \Gamma_S; \Gamma_U}$$

$$\frac{\begin{array}{l}\vdash_{\text{hind}} H_{XS}; H_{0S}; H_S; H_U; L; L_0; L_X; L_R; L_E : \Gamma_S; \Gamma_U; \epsilon \\ \vdash_{\text{j}} H_S H_U; e \qquad L; \cdot; \Gamma_S \Gamma_U; \cdot; \epsilon \vdash_{\text{ltyp}} e : \tau, \ell \qquad L_R \vdash_{\text{erel}} e\end{array}}{\vdash_{\text{lind}} H_{XS}; H_{0S}; H_S; H_U; L; L_0; L_X; L_R; L_E; e : \tau, \ell; \Gamma_S; \Gamma_U}$$

Figure C.1: Chapter 5 Safety-Proof Invariant

the result we need. In case SS5.8, the Typing Well-Formedness Lemma and Type Canonical Forms Lemma ensure locks(ℓ) is $\emptyset$, i, or α. In each case, induction and the definition of substitution through effects suffices to derive the result we need.

Lemma C.11 (Return Preservation).
If $\vdash_{\text{ret}} s$ *and* $H; \overline{L}; s \xrightarrow{\text{s}} H'; \overline{L}'; s_{\text{opt}}; s'$, *then* $\vdash_{\text{ret}} s'$.

Lemma C.12 (Values Effectless).

1. *If* $L \vdash_{\text{erel}} v$ *or* $L \vdash_{\text{erel}} x$, *then* $L = \cdot$.
2. *If* $L; \Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} v : \tau$ *and* $L; \Delta \vdash_{\text{wf}} \epsilon'$, *then* $L; \Delta; \Gamma; \gamma; \epsilon' \vdash_{\text{rtyp}} v : \tau$.
   If $L; \Delta; \Gamma; \gamma; \epsilon \vdash_{\text{ltyp}} x : \tau, \ell$ *and* $L; \Delta \vdash_{\text{wf}} \epsilon'$, *then* $L; \Delta; \Gamma; \gamma; \epsilon' \vdash_{\text{ltyp}} x : \tau, \ell$.
3. $x, v \not\downarrow_{\text{e}} v'$ *and* $x, v \not\downarrow_{\text{e}} x'$

Lemma C.13 (Access Control). *Suppose:*

1. $\vdash_{\text{hind}} H_{XS}; H_{0S}; H_S; H_U; L; L_0; L_X; L_R; L_E : \Gamma_S; \Gamma_U; \epsilon$
2. $(\Gamma_S \Gamma_U)(x) = (\tau, \ell)$

3. $\cdot; \epsilon \vdash_{\text{acc}} \ell$

Then $x \in \text{Dom}(H_S H_U)$

Proof: From the $\vdash_{\text{hind}}$ derivation and the Heap-Type Well-Formedness Lemma, $L; \cdot \vdash_{\text{wf}} \Gamma_S$ and $L; \cdot \vdash_{\text{wf}} \Gamma_U$, so the second assumption ensures $L; \cdot \vdash_{\text{k}} \ell : \texttt{LU}$. Therefore, the Type Canonical Forms Lemma ensures $\ell = loc$ or $\ell = \text{S}(i)$ for some i. The derivation of the first assumption provides $L \vdash_{\text{shr}} \Gamma_S$, so if $\ell = loc$, then $x \in \text{Dom}(\Gamma_U)$. In this case, the derivation of the first assumption provides $L; \Gamma_S \Gamma_U \vdash_{\text{htyp}} H_U : \Gamma_U$, so the Heap-Type Well-Formedness Lemma ensures $x \in \text{Dom}(H_U)$.

If $\ell = \text{S}(i)$, then $\cdot; \epsilon \vdash_{\text{acc}} \text{S}(i)$ ensures $i \in \epsilon$, which by the $\vdash_{\text{hind}}$ derivation ensures $i \in L_E$. More importantly, $i \notin L_0$ and $i \notin L_X$. From the $\vdash_{\text{hlk}}$ assumptions, that means $x \notin \text{Dom}(H_{XS})$ and $x \notin \text{Dom}(H_{0S})$. So from the Heap-Type Well-Formedness Lemma and $x \in \text{Dom}(\Gamma_S \Gamma_U)$, we conclude $x \in \text{Dom}(H_S H_U)$.

Lemma C.14 (Canonical Forms). *Suppose* $L; \cdot; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} v : \tau$.

1. *If* $\tau = \text{int}$, *then* $v = i$ *for some* i.
2. *If* $\tau = \tau_0 \times \tau_1$, *then* $v = (v_0, v_1)$ *for some* v_0 *and* v_1.
3. *If* $\tau = \tau_1 \xrightarrow{\epsilon'} \tau_2$, *then* $v = (\tau_1, \ell\ x) \xrightarrow{\epsilon'} \tau_2\ s$ *for some* ℓ, x, *and* s.
4. *If* $\tau = \tau' * \ell$, *then* $v = \&x$ *for some* x.
5. *If* $\tau = \forall \alpha{:}\kappa[\gamma'].\tau'$, *then* $v = \Lambda \alpha{:}\kappa[\gamma'].f$ *for some* f.
6. *If* $\tau = \exists \alpha{:}\kappa[\gamma'].\tau'$, *then* $v = \mathsf{pack}\ \tau'', v'\ \mathsf{as}\ \exists \alpha{:}\kappa[\gamma'].\tau'$ *for some* τ'' *and* v'.
7. *If* $\tau = \text{lock}(loc)$, *then* $v = \mathsf{nonlock}$.
8. *If* $\tau = \text{lock}(\text{S}(i))$, *then* $v = \mathsf{lock}\ i$.

Lemma C.15 (Sharable Values Need Only Sharable Context). *Suppose:*

1. $L \vdash_{\text{shr}} \Gamma_S$ *and* $L \vdash_{\text{loc}} \Gamma_U$
2. $L; \Delta; \Gamma_S \Gamma_U; \gamma; \epsilon \vdash_{\text{rtyp}} v : \tau$
3. $L; \Delta \vdash_{\text{k}} \tau : \texttt{AS}$

Then $L; \Delta; \Gamma_S; \gamma; \epsilon \vdash_{\text{rtyp}} v : \tau$

Proof: The proof is by induction on the structure of v. (Technically, several cases also need the fact that any part of a well-formed Γ is well-formed to ensure $\vdash_{\text{wf}} L; \Delta; \Gamma_S; \gamma; \epsilon$.)

- If $v = i$, SR5.5 ensures the result.
- If $v = \&x$, inverting the typing assumption ensures $\tau = \tau' * \ell$ and $(\Gamma_S \Gamma_U)(x) = (\tau', \ell)$. The third assumption and the Type Canonical Forms Lemma ensure $L; \Delta \vdash_{\mathrm{k}} \ell : \mathtt{LS}$ and $L; \Delta \vdash_{\mathrm{k}} \tau : \mathtt{AS}$. Hence $L \vdash_{\mathrm{loc}} \Gamma_U$ ensures $x \notin \Gamma_U$. So $x \in \Gamma_S$, from which we can derive the desired result.
- If $v = (\tau_1, \ell\ x) \xrightarrow{\epsilon'} \tau_2\ s$, inverting the typing assumption ensures $L; \Delta; \Gamma_1, x{:}(\tau_1, \ell); \epsilon'; \gamma; \tau_2 \vdash_{\mathrm{styp}} s$ for some Γ_1 such that $\Gamma = \Gamma_1\Gamma_2$ and $L \vdash_{\mathrm{shr}} \Gamma_1$. Because $L \vdash_{\mathrm{loc}} \Gamma_U$, we can show $\Gamma_S = \Gamma_1\Gamma'$ for some Γ'. (Technically, the proof is by induction on the size of Γ.) So the Context Weakening Lemma suffices to derive the desired result.
- If $v = \Lambda\alpha{:}\kappa[\gamma'].f$, the result follows from induction (extending Δ and γ) and the static semantics.
- If $v = (v_0, v_1)$, the result follows from induction and the static semantics.
- If $v = \mathsf{pack}\ \tau_1, v'\ \mathsf{as}\ T_2$, the result follows from induction and the static semantics.
- If $v = \mathsf{nonlock}$, SR5.17 ensures the result.
- If $v = \mathsf{lock}\ i$, SR5.15 ensures the result.

Lemma C.16 (Type and Release Preservation). *Suppose:*

1. $H_{XS}H_{XU}H_{0S}H_SH_U; (L; L_0; L_RL_E); s \xrightarrow{\mathrm{s}} H'; (L'; L'_0; L'_h); s_{\mathrm{opt}}; s'$
   (respectively, $H_{XS}H_{XU}H_{0S}H_SH_U; (L; L_0; L_RL_E); e \xrightarrow{\mathrm{r}} H'; (L'; L'_0; L'_h); s_{\mathrm{opt}}; e'$*)*
   (respectively, $H_{XS}H_{XU}H_{0S}H_SH_U; (L; L_0; L_RL_E); e \xrightarrow{\mathrm{l}} H'; (L'; L'_0; L'_h); s_{\mathrm{opt}}; e'$*)*
2. $\vdash_{\mathrm{sind}} H_{XS}; H_{0S}; H_S; H_U; L; L_0; L_X; L_R; L_E; \tau; s : \Gamma_S; \Gamma_U$
   (respectively, $\vdash_{\mathrm{rind}} H_{XS}; H_{0S}; H_S; H_U; L; L_0; L_X; L_R; L_E; e : \tau; \Gamma_S; \Gamma_U$*)*
   (respectively, $\vdash_{\mathrm{lind}} H_{XS}; H_{0S}; H_S; H_U; L; L_0; L_X; L_R; L_E; e : \tau, \ell; \Gamma_S; \Gamma_U$*)*

Then there exist H'_{XS}, H'_{0S}, H'_S, H'_U, Γ'_S, Γ'_U, L'_0, *and* L'_R *such that:*

1. $H' = H'_{XS}H_{XU}H'_{0S}H'_SH'_U$
2. $L'_h = L'_RL_E$
3. $\vdash_{\mathrm{sind}} H'_{XS}; H'_{0S}; H'_S; H'_U; L'; L'_0; L_X; L'_R; L_E; \tau; s' : \Gamma'_S; \Gamma'_U$
   (respectively, $\vdash_{\mathrm{rind}} H'_{XS}; H'_{0S}; H'_S; H'_U; L'; L'_0; L_X; L'_R; L_E; e' : \tau; \Gamma'_S; \Gamma'_U$*)*
   (respectively, $\vdash_{\mathrm{lind}} H'_{XS}; H'_{0S}; H'_S; H'_U; L'; L'_0; L_X; L'_R; L_E; e' : \tau, \ell; \Gamma'_S; \Gamma'_U$*)*

4. $L' = LL''$ *for some* L''

5. $\Gamma'_S = \Gamma_S\Gamma$ *for some* Γ *and* $\Gamma'_U = \Gamma_U\Gamma$ *for some (other)* Γ

6. $H'_{XS} = H_{XS}$ *or* $H'_{XS} = H_{XS}, x \mapsto v$ *for some* x *and* v

7. *if* $s_{\text{opt}} \neq \cdot$, *then*

   (a) $\vdash_{\text{ret}} s_{\text{opt}}$

   (b) $\cdot \vdash_{\text{srel}} s_{\text{opt}}$

   (c) $\vdash_{\text{jf}} s_{\text{opt}}$

   (d) $L'; \cdot; \Gamma'_S; \cdot; \emptyset; \tau' \vdash_{\text{styp}} s_{\text{opt}}$ *for some* τ'

Proof: The proofs are by simultaneous induction on the derivations of the dynamic step, proceeding by cases on the last step in the derivation. Throughout, let $H_j = H_S H_U$, $C = L; \cdot; \Gamma_S\Gamma_U; \cdot; \epsilon$, $H'_j = H'_S H'_U$, and $C' = L'; \cdot; G'_S\Gamma'_U; \cdot; \epsilon$.

If the heap does not change, Conclusions 1, 5, and 6 are trivial by letting $H'_{XS} = H_{XS}$, $H'_{0S} = H_{0S}$, $H'_S = H_S$, $H'_U = H_U$, $\Gamma'_S = \Gamma_S$, and $\Gamma'_U = \Gamma_U$. If no lock collection changes, Conclusions 2 and 4 are trivial by letting $L' = L$, $L'_0 = L_0$, and $L'_R = L_R$. If $s_{\text{opt}} = \cdot$, Conclusion 7 holds vacuously. If the heap does not change, no lock collection changes, and $s_{\text{opt}} = \cdot$, we the case is <u>local</u>. For a local case, only Conclusion 3 remains and the $\vdash_{\text{hind}}$ conclusion we need is provided by inverting the typing assumption. Hence it suffices to show:
$\vdash_{\text{j}} H_j; s'$, $L_R \vdash_{\text{srel}} s'$, and $C; \tau \vdash_{\text{styp}} s'$
(respectively, $\vdash_{\text{j}} H_j; e'$, $L_R \vdash_{\text{erel}} e'$, and $C \vdash_{\text{rtyp}} e' : \tau$)
(respectively, $\vdash_{\text{j}} H_j; e'$, $L_R \vdash_{\text{erel}} e'$, and $C \vdash_{\text{ltyp}} e' : \tau, \ell$).
Inverting the $\vdash_{\text{hind}}$ assumption, we can assume
$\vdash_{\text{j}} H_j; s$, $L_R \vdash_{\text{srel}} s$, and $C; \tau \vdash_{\text{styp}} s$
(respectively, $\vdash_{\text{j}} H_j; e$, $L_R \vdash_{\text{erel}} e$, and $C \vdash_{\text{rtyp}} e : \tau$)
(respectively, $\vdash_{\text{j}} H_j; e$, $L_R \vdash_{\text{erel}} e$, and $C \vdash_{\text{ltyp}} e : \tau, \ell$). Using these assumptions, we derive our three obligations (underlining them in each case).

Most of the <u>inductive</u> cases follow a similar form: To invoke the induction hypothesis, we invert the $\vdash_{\text{sind}}$ (respectively $\vdash_{\text{rind}}$ or $\vdash_{\text{lind}}$) assumption to provide a $\vdash_{\text{hind}}$ assumption, a type-checking assumption, a release assumption, and a junk assumption. For the induction hypothesis to apply, we use the $\vdash_{\text{hind}}$ assumption unchanged and invert the other assumptions to get the facts we need. We then use the result of the induction to provide the result we need: Using the H'_{XS}, H'_{0S}, H'_S, H'_U, Γ'_S, Γ'_U, L'_0, and L'_R from the result of the induction, only Conclusion 3 remains and the $\vdash_{\text{hind}}$ conclusion we need is provided by inverting Conclusion 3 from the induction. We use the other assumptions from this inversion to derive

the other facts we need to derive Conclusion 3. Hence for inductive cases, we just explain what facts we use to invoke the induction hypothesis, and how we use the result to derive the facts we need for Conclusion 3. In each case, we underline these facts.

- DL5.1: Let $e = *\&x$ and $e' = x$. The case is local. Because $x', v' \not\vdash_{\mathrm{e}} x$, inverting $\vdash_{\mathrm{j}} H_j; *\&x$ ensures $\vdash_{\mathrm{jf}} H_j$, so we can derive $\underline{\vdash_{\mathrm{j}} H_j; x}$. Inverting $L_R \vdash_{\mathrm{erel}} *\&x$ ensures $\underline{L_R \vdash_{\mathrm{erel}} x}$. Inverting $C \vdash_{\mathrm{ltyp}} *\&x : \tau, \ell$ ensures $\underline{C \vdash_{\mathrm{ltyp}} x : \tau, \ell}$.

- DL5.2: Let $e = *e_1$ and $e' = *e'_1$. The case is inductive. Inverting $\vdash_{\mathrm{j}} H_j; *e_1$ ensures $\vdash_{\mathrm{j}} H_j; e_1$. Inverting $L_R \vdash_{\mathrm{erel}} *e_1$ ensures $L_R \vdash_{\mathrm{erel}} *e_1$. Inverting $C \vdash_{\mathrm{ltyp}} *e_1 : \tau, \ell$ ensures $C \vdash_{\mathrm{rtyp}} e_1 : \tau*\ell$. So the induction hypothesis provides $\vdash_{\mathrm{j}} H'_j; e'_1$ (so $\underline{\vdash_{\mathrm{j}} H'_j; *e'_1}$), $L'_R \vdash_{\mathrm{erel}} e'_1$ (so $\underline{L'_R \vdash_{\mathrm{erel}} *e'_1}$), and $C' \vdash_{\mathrm{rtyp}} e'_1 : \tau*\ell$ (so $\underline{C' \vdash_{\mathrm{ltyp}} *e'_1 : \tau, \ell}$).

- DR5.1: Let $e = x$ and $e' = H(x)$. The case is local. Inverting $\vdash_{\mathrm{j}} H_j; x$ ensures $\vdash_{\mathrm{jf}} H_j$. Inverting $L_R \vdash_{\mathrm{erel}} x$ ensures $L_R = \cdot$. Inverting $C \vdash_{\mathrm{rtyp}} x : \tau$ ensures $(\Gamma_S\Gamma_U)(x) = (\tau, \ell)$, $\cdot; \epsilon \vdash_{\mathrm{acc}} \ell$, and $\vdash_{\mathrm{wf}} C$. So the Access Control Lemma ensures $x \in \mathrm{Dom}(H_j)$. Therefore, $\vdash_{\mathrm{jf}} H(x)$, so $\underline{\vdash_{\mathrm{j}} H_j; H(x)}$. Because the $\vdash_{\mathrm{hind}}$ assumptions ensure H_S and H_U are well-typed and $x \in \mathrm{Dom}(H_j)$, the Heap-Type Well-Formedness Lemma ensures $\underline{\cdot \vdash_{\mathrm{erel}} H(x)}$ and either $L; \cdot; \Gamma_S; \cdot; \emptyset \vdash_{\mathrm{rtyp}} H(x) : \tau$ or $L; \cdot; \Gamma_S\Gamma_U; \cdot; \emptyset \vdash_{\mathrm{rtyp}} H(x) : \tau$. In either case, the Term Weakening Lemma ensures $\underline{C \vdash_{\mathrm{rtyp}} H(x) : \tau}$.

- DR5.2A: Let $e = x{=}v$ and $e' = (x{=}\mathsf{junk}_v)$. Inverting the $\vdash_{\mathrm{rind}}$ assumption ensures $\vdash_{\mathrm{j}} H_j; x{=}v$, $L_R \vdash_{\mathrm{erel}} x{=}v$, and $C \vdash_{\mathrm{rtyp}} x{=}v : \tau$. Inverting $C \vdash_{\mathrm{rtyp}} x{=}v : \tau$ ensures $(\Gamma_S\Gamma_U)(x) = (\tau, \ell)$, $\vdash_{\mathrm{wf}} C$, $C \vdash_{\mathrm{rtyp}} v : \tau$, and $\cdot; \epsilon \vdash_{\mathrm{acc}} \ell$. So the Access Control Lemma ensures $x \in \mathrm{Dom}(H_j)$. Therefore, either $H_S = H_1, x \mapsto v'$ or $H_U = H_1, x \mapsto v'$. In the former case, let $H'_S = H_1, x \mapsto \mathsf{junk}_v$ and $H'_U = H_U$; in the latter case, let $H'_S = H_S$ and $H'_U = H_1, x \mapsto \mathsf{junk}_v$. Letting $H'_{XS} = H_{XS}$, $H'_{0S} = H_{0S}$, $L'_0 = L_0$, $L'_R = L_R$, $\Gamma'_S = \Gamma_S$, $\Gamma'_U = \Gamma_U$, and $s_{\mathrm{opt}} = \cdot$, all of the conclusions follow immediately from the assumptions, except for Conclusion 3.

  First we show $\vdash_{\mathrm{hind}} H_{XS}; H_{0S}; H'_S; H'_U; L; L_0; L_X; L_R; L_E; e' : \tau; \Gamma_S; \Gamma_U; \epsilon$.
  If $x \in \mathrm{Dom}(H_U)$, then the $\vdash_{\mathrm{hind}}$ derivation in the assumptions provides all of the hypotheses except for $L; \Gamma_S\Gamma_U \vdash_{\mathrm{htyp}} H'_U : \Gamma_U$. Inverting $L; \Gamma_S\Gamma_U \vdash_{\mathrm{htyp}} H_U : \Gamma_U$ provides $\underline{L; \Gamma_S\Gamma_U \vdash_{\mathrm{htyp}} H_1 : \Gamma_1}$ where $\Gamma_U = \Gamma_1, x{:}(\tau, \ell)$ for some Γ_1. Given $C \vdash_{\mathrm{rtyp}} v : \tau$, the Values Effectless Lemma ensures $L; \cdot; \Gamma_S\Gamma_U; \cdot; \emptyset \vdash_{\mathrm{rtyp}} v : \tau$, so $\underline{L; \cdot; \Gamma_S\Gamma_U; \cdot; \emptyset \vdash_{\mathrm{rtyp}} \mathsf{junk}_v : \tau}$. Inverting $L_R \vdash_{\mathrm{erel}} x{=}v$ ensures $L_R \vdash_{\mathrm{erel}} v$, so the Values Effectless Lemma ensures $L_R = \cdot$. So $\underline{\cdot \vdash_{\mathrm{erel}} \mathsf{junk}_v}$. The underlined facts let us derive $L; \Gamma_S\Gamma_U \vdash_{\mathrm{htyp}} H'_U : \Gamma_U$.

If $x \in \mathrm{Dom}(H_S)$, then the $\vdash_{\mathrm{hind}}$ derivation in the assumptions provides all of the hypotheses except for $L; \Gamma_S \vdash_{\mathrm{htyp}} H_{XS} H_{0S} H'_S : \Gamma_S$ and $\Gamma_S; L_R L_E \vdash_{\mathrm{hlk}} H'_S$. The latter follows from the derivation of $\Gamma_S; L_R L_E \vdash_{\mathrm{hlk}} H_S$ because $\Gamma'_S = \Gamma_S$. For the former, inverting $L; \Gamma_S \vdash_{\mathrm{htyp}} H_{XS} H_{0S} H_S : \Gamma_S$ provides $\underline{L; \Gamma_S \vdash_{\mathrm{htyp}} H_{XS} H_{0S} H_1 : \Gamma_1}$ where $\Gamma_S = \Gamma_1, x{:}(\tau, \ell)$ for some Γ_1. Given $C \vdash_{\mathrm{rtyp}} v : \tau$, the Values Effectless Lemma ensures $L; \cdot; \Gamma_S \Gamma_U; \cdot; \emptyset \vdash_{\mathrm{rtyp}} v : \tau$, so $L; \cdot; \Gamma_S \Gamma_U; \cdot; \emptyset \vdash_{\mathrm{rtyp}} \mathsf{junk}_v : \tau$. Because $L \vdash_{\mathrm{shr}} \Gamma_S$ and $\Gamma_S(x) = (\tau, \ell)$, we know $L; \cdot \vdash_{\mathrm{k}} \tau : \mathtt{AS}$. Therefore, the Sharable Values Need Only Sharable Context Lemma ensures $\underline{L; \cdot; \Gamma_S; \cdot; \emptyset \vdash_{\mathrm{rtyp}} \mathsf{junk}_v : \tau}$. Inverting $L_R \vdash_{\mathrm{erel}} x{=}v$ ensures $L_R \vdash_{\mathrm{erel}} v$, so the Values Effectless Lemma ensures $L_R = \cdot$. So $\underline{\cdot \vdash_{\mathrm{erel}} \mathsf{junk}_v}$. The underlined facts let us derive $L; \Gamma_S \vdash_{\mathrm{htyp}} H_{XS} H_{0S} H'_S : \Gamma_S$.

To conclude $\vdash_{\mathrm{rind}} H_{XS}; H_{0S}; H'_S; H'_U; L; L_0; L_X; L_R; L_E; (x{=}\mathsf{junk}_v) : \tau; \Gamma_S; \Gamma_U$, we still must show $\vdash_{\mathrm{j}} H'_j; x{=}\mathsf{junk}_v$, $C \vdash_{\mathrm{rtyp}} x{=}\mathsf{junk}_v : \tau$, and $L_R \vdash_{\mathrm{erel}} x{=}\mathsf{junk}_v$. Inverting $\vdash_{\mathrm{j}} H_j; x{=}v$, the Values Effectless Lemma ensures $\vdash_{\mathrm{jf}} H_j$ and $\vdash_{\mathrm{jf}} v$, from which we can derive $\vdash_{\mathrm{j}} H'_j; x{=}\mathsf{junk}_v$ (because $H'_j(x) = \mathsf{junk}_v$ and H'_j is otherwise junk-free). From $C \vdash_{\mathrm{rtyp}} v : \tau$, we derive $C \vdash_{\mathrm{rtyp}} \mathsf{junk}_v : \tau$, so with the other facts from inverting $C \vdash_{\mathrm{rtyp}} x{=}v : \tau$ (see above), we can derive $C \vdash_{\mathrm{rtyp}} x{=}\mathsf{junk}_v : \tau$. Finally, for $x \in \mathrm{Dom}(H_S)$ or $x \in \mathrm{Dom}(H_U)$, we showed $L_R = \cdot$ and $\cdot \vdash_{\mathrm{erel}} v$, so we can derive $L_R \vdash_{\mathrm{erel}} x{=}\mathsf{junk}_v$.

- DR5.2B: Let $e = (x{=}\mathsf{junk}_v)$ and $e' = v$. Inverting the $\vdash_{\mathrm{rind}}$ assumption ensures $\vdash_{\mathrm{j}} H_j; x{=}\mathsf{junk}_v$, $L_R \vdash_{\mathrm{erel}} x{=}\mathsf{junk}_v$, and $C \vdash_{\mathrm{rtyp}} x{=}\mathsf{junk}_v : \tau$. Inverting $\vdash_{\mathrm{j}} H_j; x{=}\mathsf{junk}_v$ ensures $H_j = H_1, x \mapsto \mathsf{junk}_v$, $\vdash_{\mathrm{jf}} H_1$, and $\vdash_{\mathrm{jf}} v$ for some H_1. So either $H_S = H_2, x \mapsto \mathsf{junk}_v$ or $H_U = H_2, x \mapsto \mathsf{junk}_v$ for some H_2. In the former case, let $H'_S = H_2, x \mapsto v$ and $H'_U = H_U$; in the latter case, let $H'_S = H_S$ and $H'_U = H_2, x \mapsto v$. Letting $H'_{XS} = H_{XS}$, $H'_{0S} = H_{0S}$, $L'_0 = L_0$, $L'_R = L_R$, $\Gamma'_S = \Gamma_S$, $\Gamma'_U = \Gamma_U$, and $s_{\mathrm{opt}} = \cdot$, all of the conclusions follow immediately from the assumptions, except for Conclusion 3.

  First we show $\vdash_{\mathrm{hind}} H_{XS}; H_{0S}; H'_S; H'_U; L; L_0; L_X; L_R; L_E : \Gamma_S; \Gamma_U; \epsilon$. If $x \in \mathrm{Dom}(H_U)$, then the $\vdash_{\mathrm{hind}}$ derivation in the assumptions provides all of the hypotheses except for $L; \Gamma_S \vdash_{\mathrm{htyp}} \Gamma_U : H'_U \Gamma_U$. Inverting $L; \Gamma_S \Gamma_U \vdash_{\mathrm{htyp}} H_U : \Gamma_U$ provides $\underline{L; \Gamma_S \Gamma_U \vdash_{\mathrm{htyp}} H_1 : \Gamma_1}$ where $\Gamma_U = \Gamma_1, x{:}(\tau, \ell)$ for some Γ_1, $\cdot \vdash_{\mathrm{erel}} \mathsf{junk}_v$, and $L; \cdot; \overline{\Gamma_S \Gamma_U; \cdot; \emptyset \vdash_{\mathrm{rtyp}} \mathsf{junk}_v : \tau}$. Inverting the typing and release results ensures $\underline{L; \cdot; \Gamma_S \Gamma_U; \cdot; \emptyset \vdash_{\mathrm{rtyp}} v : \tau}$ and $\underline{\cdot \vdash_{\mathrm{erel}} v}$. The underlined facts let us derive $L; \Gamma_S \Gamma_U \vdash_{\mathrm{htyp}} H'_U : \Gamma_U$.

  If $x \in \mathrm{Dom}(H_S)$, the argument is analogous (using Γ_S in place of $\Gamma_S \Gamma_U$), but we also must show $\Gamma_S; L_R L_E \vdash_{\mathrm{hlk}} H'_S$, which follows from the derivation of $\Gamma_S; L_R L_E \vdash_{\mathrm{hlk}} H_S$ because $\Gamma'_S = \Gamma_S$.

  To conclude $\vdash_{\mathrm{rind}} H_{XS}; H_{0S}; H'_S; H'_U; L; L_0; L_X; L_R; L_E; v : \tau; \Gamma_S; \Gamma_U$, we still

must show $\vdash_{\mathrm{j}} H'_j; v$, $C \vdash_{\mathrm{rtyp}} v : \tau$, and $L_R \vdash_{\mathrm{erel}} v$. The latter two follow from inversion of $C \vdash_{\mathrm{rtyp}} x{=}\mathsf{junk}_v : \tau$ and $L_R \vdash_{\mathrm{erel}} x{=}\mathsf{junk}_v$. We showed above that $H_j = H_1, x \mapsto \mathsf{junk}_v$ for some H_1 for which $\vdash_{\mathrm{jf}} H_1$ and $\vdash_{\mathrm{jf}} v$. So $H'_j = H_1, x \mapsto v$ and we can derive $\vdash_{\mathrm{j}} H'_j; v$.

- DR5.3: Let $e = *\&x$ and $e' = x$. This case is local. Because $x', v' \not\vdash_{\mathrm{e}} x$, inverting $\vdash_{\mathrm{j}} H_j; *\&x$ ensures $\vdash_{\mathrm{jf}} H_j$, so we can derive $\underline{\vdash_{\mathrm{j}} H_j; x}$. Inverting $L_R \vdash_{\mathrm{erel}} *\&x$ ensures $\underline{L_R \vdash_{\mathrm{erel}} x}$. Inverting $C \vdash_{\mathrm{rtyp}} *\&x : \tau$ ensures $(\Gamma_S\Gamma_U)(x) = (\tau, \ell)$, $\cdot; \epsilon \vdash_{\mathrm{acc}} \ell$, and $\vdash_{\mathrm{wf}} C$, so we can derive $\underline{C \vdash_{\mathrm{rtyp}} x : \tau}$.

- DR5.4: Let $e = (v_0, v_1).i$ and $e' = v_i$. This case is local. We assume $i = 0$; the argument is analogous if $i = 1$. By the Values Effectless Lemma, $x', v' \not\vdash_{\mathrm{e}} (v_0, v_1)$, so inverting $\vdash_{\mathrm{j}} H_j; (v_0, v_1).i$ ensures $\vdash_{\mathrm{jf}} H_j$ and $\vdash_{\mathrm{jf}} v_0$. So we can derive $\underline{\vdash_{\mathrm{j}} H_j; v_0}$. By the Values Effectless Lemma, if $L_R \vdash_{\mathrm{erel}} (v_0, v_1)$, then $L_R = \cdot$, so inverting $L_R \vdash_{\mathrm{erel}} (v_0, v_1).i$ ensures $\underline{L_R \vdash_{\mathrm{erel}} v.0}$ (because $L_R = \cdot$, it does not matter which rule derives $L_R \vdash_{\mathrm{erel}} (v_0, v_1)$). Inverting $C \vdash_{\mathrm{rtyp}} (v_0, v_1).i : \tau$ ensures $C \vdash_{\mathrm{rtyp}} v_0 : \tau_0$.

- DR5.5: Let $e = ((\tau_1, \ell\ x) \xrightarrow{\epsilon'} \tau_2\ s)(v)$ and $e' = \mathsf{call}\ (\mathsf{let}\ \ell, x{=}v;\ s)$. The case is local. The Values Effectless Lemma and inverting $\vdash_{\mathrm{j}} H_j; ((\tau_1, \ell\ x) \xrightarrow{\epsilon'} \tau_2\ s)(v)$ ensure $\vdash_{\mathrm{jf}} H_j$, $\vdash_{\mathrm{jf}} s$, and $\vdash_{\mathrm{jf}} v$. So we can derive $\vdash_{\mathrm{jf}} \mathsf{call}\ (\mathsf{let}\ \ell, x{=}v;\ s)$ and therefore $\underline{\vdash_{\mathrm{j}} H; \mathsf{call}\ (\mathsf{let}\ \ell, x{=}v;\ s)}$. The Values Effectless Lemma and inverting $L_R \vdash_{\mathrm{erel}} ((\tau_1, \ell\ x) \xrightarrow{\epsilon'} \tau\ s)(v)$ ensures $L_R = \cdot$, $\cdot \vdash_{\mathrm{erel}} v$, and $\cdot \vdash_{\mathrm{srel}} s$. So we can derive $\underline{\cdot \vdash_{\mathrm{erel}} \mathsf{call}\ (\mathsf{let}\ \ell, x{=}v;\ s)}$. Inverting $C \vdash_{\mathrm{rtyp}} ((\tau_1, \ell\ x) \xrightarrow{\epsilon'} \tau_2\ s)(v) : \tau$ ensures $\tau_2 = \tau$, $L; \cdot; \Gamma_1, x{:}(\tau_1, \ell); \cdot; \epsilon'; \tau \vdash_{\mathrm{styp}} s$ (where $\Gamma_S\Gamma_U = \Gamma_1\Gamma_2$ for some Γ_2), $\vdash_{\mathrm{ret}} s$, $C \vdash_{\mathrm{rtyp}} v : \tau_1$, and $\cdot \vdash_{\mathrm{eff}} \epsilon' \subseteq \epsilon$. So by the Context Weakening Lemma $L; \cdot; \Gamma_S\Gamma_U, x{:}(\tau_1, \ell); \cdot; \epsilon; \tau \vdash_{\mathrm{styp}} s$. So with SS5.6 and SR5.10 we can derive $\underline{C \vdash_{\mathrm{rtyp}} \mathsf{call}\ (\mathsf{let}\ \ell, x{=}v;\ s) : \tau}$.

- DR5.6: Let $e = \mathsf{call}\ \mathsf{return}\ v$ and $e' = v$. The case is local. Inverting $\vdash_{\mathrm{j}} H_j; \mathsf{call}\ \mathsf{return}\ v$ ensures $\underline{\vdash_{\mathrm{j}} H_j; v}$. Inverting $L_R \vdash_{\mathrm{erel}} \mathsf{call}\ \mathsf{return}\ v$ ensures $\underline{L_R \vdash_{\mathrm{erel}} v}$. Inverting $C \vdash_{\mathrm{rtyp}} \mathsf{call}\ \mathsf{return}\ v : \tau$ ensures $\underline{C \vdash_{\mathrm{rtyp}} v : \tau}$.

- DR5.7: Let $e = (\Lambda\alpha{:}\kappa[\gamma].f)[\tau']$ and $e' = f[\tau'/\alpha]$. The case is local. The Values Effectless Lemma and inverting $\vdash_{\mathrm{j}} H_j; (\Lambda\alpha{:}\kappa[\gamma].f)[\tau']$ ensure $\vdash_{\mathrm{jf}} H_j$ and $\vdash_{\mathrm{jf}} f$. So Term Substitution Lemma 3 ensures $\vdash_{\mathrm{jf}} f[\tau'/\alpha]$ and therefore $\underline{\vdash_{\mathrm{j}} H_j; f[\tau'/\alpha]}$. The Values Effectless Lemma and inverting $\underline{L_R \vdash_{\mathrm{erel}} (\Lambda\alpha{:}\kappa[\gamma].f)[\tau']}$ ensure $L_R \vdash_{\mathrm{erel}} f$. So Term Substitution Lemma 2 ensures $L_R \vdash_{\mathrm{erel}} f[\tau'/\alpha]$. Inverting $C \vdash_{\mathrm{rtyp}} \Lambda\alpha{:}\kappa[\gamma].f : \forall\alpha{:}\kappa[\gamma].\tau''$ ensures $\tau = \tau''[\tau'/\alpha]$, $L; \alpha{:}\kappa; \Gamma_S\Gamma_U; \gamma; \epsilon \vdash_{\mathrm{rtyp}} f : \tau''$, $\vdash_{\mathrm{wf}} C$, $L; \cdot \vdash_{\mathrm{k}} \forall\alpha{:}\kappa[\gamma].\tau'' : \mathtt{AU}$,

$L;\cdot \vdash_{\text{k}} \tau' : \kappa$, and $\cdot \vdash_{\text{eff}} \gamma[\tau'/\alpha]$. So Term Substitution Lemma 4 ensures $L;\cdot;(\Gamma_S\Gamma_U)[\tau'/\alpha];\gamma[\tau'/\alpha];\epsilon[\tau'/\alpha] \vdash_{\text{rtyp}} f[\tau'/\alpha] : \tau''[\tau'/\alpha]$. So the Useless Substitution Lemma ensures $L;\cdot;\Gamma_S\Gamma_U;\gamma[\tau'/\alpha];\epsilon \vdash_{\text{rtyp}} f[\tau'/\alpha] : \tau''[\tau'/\alpha]$. Finally, the Term Weakening Lemma ensures $\underline{L;\cdot;\Gamma_S\Gamma_U;\cdot;\epsilon \vdash_{\text{rtyp}} f[\tau'/\alpha] : \tau''[\tau'/\alpha]}$.

- DR5.8: Let $e = \mathsf{newlock}()$ and $e' = \mathsf{pack}\ \mathrm{S}(i), \mathsf{lock}\ i\ \mathsf{as}\ \exists\alpha{:}\mathtt{LS}[\cdot].\mathrm{lock}(\alpha)$. Letting $H'_{XS} = H_{XS}$, $H'_{0S} = H_{0S}$, $H'_S = H_S$, $H'_U = H_U$, $\Gamma'_S = \Gamma_S$, $\Gamma'_U = \Gamma_U$, $\underline{L'_0 = L_0, i}$, and $\ell'_R = L_R$, all of the conclusions follow immediately except for Conclusion 3.

  First we show $\vdash_{\text{hind}} H_{XS};H_{0S};H_S;H_U;L';L'_0;L_X;L_R;L_E : \Gamma_S;\Gamma_U;\epsilon$. By our choice of L'_0, we know $L' = L'_0L_XL_RL_E$ (because DR5.8 ensures $L' = L,i$). The other obligations follow from inversion of $\vdash_{\text{hind}} H_{XS};H_{0S};H_S;H_U;L;L_0;L_X;L_R;L_E : \Gamma_S;\Gamma_U;\epsilon$, the Context Weakening Lemma, and the Heap-Type Weakening Lemma.

  To conclude $\vdash_{\text{rind}} H_{XS};H_{0S};H_S;H_U;L';L'_0;L_X;L_R;L_E;e' : \tau;\Gamma_S;\Gamma_U$, we still must show $\vdash_{\text{j}} H_j;e'$, $C \vdash_{\text{rtyp}} e' : \tau$, and $L_R \vdash_{\text{erel}} e'$. Inverting $\vdash_{\text{j}} H_j;\mathsf{newlock}()$ ensures $\vdash_{\text{jf}} H_j$. Because $\vdash_{\text{jf}} e'$, we conclude $\vdash_{\text{j}} H_j;e'$. Inverting $L_R \vdash_{\text{erel}} \mathsf{newlock}()$ ensures $L_R = \cdot$, so we can derive $L_R \vdash_{\text{erel}} e'$. Inverting $C \vdash_{\text{rtyp}} \mathsf{newlock}() : \tau$ ensures $\tau = \exists\alpha{:}\mathtt{LS}[\cdot].\mathrm{lock}(\alpha)$ and $\vdash_{\text{wf}} C$. So we can derive our last obligation as follows (note $C' = \ell';\cdot;\Gamma_S\Gamma_U;\cdot;\epsilon$ and $\vdash_{\text{wf}} C'$ follows from $\vdash_{\text{wf}} C$ and the Context Weakening Lemma):

$$\frac{\dfrac{i \in L' \quad \vdash_{\text{wf}} C'}{C' \vdash_{\text{rtyp}} \mathsf{lock}\ i : \mathrm{lock}(\mathrm{S}(i))} \quad \dfrac{i \in L'}{L';\cdot \vdash_{\text{k}} \mathrm{S}(i) : \mathtt{LS}} \quad \dfrac{}{\cdot \vdash_{\text{eff}} \cdot} \quad L';\cdot \vdash_{\text{k}} \exists\alpha{:}\mathtt{LS}[\cdot].\mathrm{lock}(\alpha) : \mathtt{AU}}{C' \vdash_{\text{rtyp}} \mathsf{pack}\ \mathrm{S}(i), \mathsf{lock}\ i\ \mathsf{as}\ \exists\alpha{:}\mathtt{LS}[\cdot].\mathrm{lock}(\alpha) : \exists\alpha{:}\mathtt{LS}[\cdot].\mathrm{lock}(\alpha)}$$

- DR5.9: Let $e = \mathsf{call}\ s$ and $e' = \mathsf{call}\ s'$. The case is inductive. Inverting $\vdash_{\text{j}} H_j;\mathsf{call}\ s$ ensures $\vdash_{\text{j}} H_j;s$. Inverting $L_R \vdash_{\text{erel}} \mathsf{call}\ s$ ensures $L_R \vdash_{\text{srel}} s$. Inverting $C \vdash_{\text{rtyp}} \mathsf{call}\ s : \tau$ ensures $C;\tau \vdash_{\text{styp}} s$ and $\vdash_{\text{ret}} s$. So the induction hypothesis provides $\vdash_{\text{j}} H'_j;s'$ (so $\underline{\vdash_{\text{j}} H'_j;\mathsf{call}\ s'}$), $L'_R \vdash_{\text{srel}} s'$ (so $\underline{L'_R \vdash_{\text{erel}} \mathsf{call}\ s'}$, and $C';\tau \vdash_{\text{styp}} s'$. The Return Preservation Lemma ensures $\vdash_{\text{ret}} s'$, so $\underline{C' \vdash_{\text{rtyp}} \mathsf{call}\ s' : \tau}$.

- DR5.10: There are two inductive cases. If $e = \&e_1$, let $e' = \&e'_1$. Inverting $\vdash_{\text{j}} H_j;\&e_1$ ensures $\vdash_{\text{j}} H_j;e_1$. Inverting $L_R \vdash_{\text{erel}} \&e_1$ ensures $L_R \vdash_{\text{erel}} \&e_1$. Inverting $C \vdash_{\text{rtyp}} \&e_1 : \tau$ ensures $\tau = \tau'{*}\ell$ and $C \vdash_{\text{ltyp}} e_1 : \tau',\ell$. So the induction hypothesis provides $\vdash_{\text{j}} H'_j;e'_1$ (so $\underline{\vdash_{\text{j}} H'_j;\&e'_1}$), $L'_R \vdash_{\text{erel}} e'_1$ (so $\underline{L'_R \vdash_{\text{erel}} \&e'_1}$) and $C' \vdash_{\text{ltyp}} e'_1 : \tau',\ell$ (so $\underline{C' \vdash_{\text{rtyp}} \&e'_1 : \tau'{*}\ell}$).

  If $e = (e_1{=}e_2)$, let $e' = (e'_1{=}e_2)$. By inspection of the dynamic semantics, e_1 is not some x. So inverting $\vdash_{\text{j}} H_j;e_1{=}e_2$ ensures $\vdash_{\text{j}} H_j;e_1$ and $\vdash_{\text{jf}} e_2$. Similarly,

$L_R \vdash_{\text{erel}} e_1$ and $\cdot \vdash_{\text{erel}} e_2$. Inverting $C \vdash_{\text{rtyp}} e_1{=}e_2 : \tau$ ensures $C \vdash_{\text{ltyp}} e_1 : \tau, \ell$, $C \vdash_{\text{rtyp}} e_2 : \tau$, and $\cdot; \epsilon \vdash_{\text{acc}} \ell$. So the induction hypothesis provides $\vdash_{\text{j}} H'_j; e'_1$ (so with $\vdash_{\text{jf}} e_2$ we have $\underline{\vdash_{\text{j}} H'_j; e'_1{=}e_2}$), $L'_R \vdash_{\text{erel}} e'_1$ (so with $\cdot \vdash_{\text{erel}} e_2$ we have $\underline{L'_R \vdash_{\text{erel}} e'_1{=}e_2}$), and $C' \vdash_{\text{ltyp}} e'_1 : \tau, \ell$. By the Term Weakening Lemma, $C' \vdash_{\text{rtyp}} e_2 : \tau$. So because $C'_\epsilon = \epsilon$, $\cdot'_\gamma = \cdot$, and $\cdot; \epsilon \vdash_{\text{acc}} \ell$, we can derive $\underline{C' \vdash_{\text{rtyp}} e'_1{=}e_2 : \tau}$.

- DR5.11: There are nine inductive cases. If $e = *e_1$, let $e' = *e'_1$. Inverting $\vdash_{\text{j}} H_j; *e_1$ ensures $\vdash_{\text{j}} H_j; e_1$. Inverting $L_R \vdash_{\text{erel}} *e_1$ ensures $L_R \vdash_{\text{erel}} e_1$. Inverting $C \vdash_{\text{rtyp}} *e_1 : \tau$ ensures $C \vdash_{\text{rtyp}} e_1 : \tau{*}\ell$ and $\cdot; \epsilon \vdash_{\text{acc}} \ell$. So the induction hypothesis provides $\vdash_{\text{j}} H'_j; e'_1$ (so $\underline{\vdash_{\text{j}} H'_j; *e'_1}$), $L'_R \vdash_{\text{erel}} e'_1$ (so $\underline{L'_R \vdash_{\text{erel}} *e'_1}$), and $C' \vdash_{\text{rtyp}} e'_1 : \tau{*}\ell$ (so $\underline{C' \vdash_{\text{rtyp}} *e'_1 : \tau}$ because $C'_\epsilon = \epsilon$, $C'_\gamma = \cdot$, and $\cdot; \epsilon \vdash_{\text{acc}} \ell$).

  If $e = e.i$, let $e' = e'_1.i$. Inverting $\vdash_{\text{j}} H_j; e_1.i$ ensures $\vdash_{\text{j}} H_j; e_1$. Inverting $L_R \vdash_{\text{erel}} e_1.i$ ensures $L_R \vdash_{\text{erel}} e_1$. Inverting $C \vdash_{\text{rtyp}} e_1.i : \tau$ ensures $\tau = \tau_i$ and $C \vdash_{\text{rtyp}} e_1 : \tau_0 \times \tau_1$. So the induction hypothesis provides $\vdash_{\text{j}} H'_j; e'_1$ (so $\underline{\vdash_{\text{j}} H'_j; e'_1.i}$), $L'_R \vdash_{\text{erel}} e'_1$ (so $\underline{L'_R \vdash_{\text{erel}} e'_1.i}$), and $C' \vdash_{\text{rtyp}} e'_1 : \tau_0 \times \tau_1$ (so $C' \vdash_{\text{rtyp}} e'_1.i : \tau_i$).

  If $e = (x{=}e_1)$, let $e' = (x{=}e'_1)$. Inverting $\vdash_{\text{j}} H_j; x{=}e_1$ ensures $\vdash_{\text{j}} H_j; e_1$ (because $x', v \not\vdash_{\text{e}} x$). Inverting $L_R \vdash_{\text{erel}} x{=}e_1$ ensures $L_R \vdash_{\text{erel}} e_1$ (because if $L_R \vdash_{\text{erel}} x$, then $L_R = \cdot$ and $\cdot \vdash_{\text{erel}} e_1$). Inverting $C \vdash_{\text{rtyp}} x{=}e_1 : \tau$ ensures $C \vdash_{\text{ltyp}} x : \tau, \ell$, $C \vdash_{\text{rtyp}} e_1 : \tau$, and $\cdot; \epsilon \vdash_{\text{acc}} \ell$. So the induction hypothesis provides $\vdash_{\text{j}} H'_j; e'_1$ (so $\underline{\vdash_{\text{j}} H'_j; x{=}e'_1}$), $L'_R \vdash_{\text{erel}} e'_1$ (so $L'_R \vdash_{\text{erel}} x{=}e'_1$), and $C' \vdash_{\text{rtyp}} e'_1 : \tau$. By the Term Weakening Lemma, $C' \vdash_{\text{ltyp}} x : \tau, \ell$. So because $C'_\epsilon = \epsilon$, $C'_\gamma = \cdot$, and $\cdot; \epsilon \vdash_{\text{acc}} \ell$, we can derive $\underline{C' \vdash_{\text{rtyp}} x{=}e'_1 : \tau}$.

  If $e = e_1[\tau']$, let $e' = e'_1[\tau']$. Inverting $\vdash_{\text{j}} H_j; e_1[\tau']$ ensures $\vdash_{\text{j}} H_j; e_1$. Inverting $L_R \vdash_{\text{erel}} e_1[\tau']$ ensures $L_R \vdash_{\text{erel}} e_1$. Inverting $C \vdash_{\text{rtyp}} e_1[\tau'] : \tau$ ensures $\tau = \tau''[\tau'/\alpha]$, $C \vdash_{\text{rtyp}} e_1 : \forall\alpha{:}\kappa[\gamma].\tau''$ $L; \cdot \vdash_{\text{k}} \tau' : \kappa$, and $\cdot \vdash_{\text{eff}} \gamma$. So the induction hypothesis provides $\vdash_{\text{j}} H'_j; e'_1$ (so $\underline{\vdash_{\text{j}} H'_j; e'_1[\tau']}$), $L'_R \vdash_{\text{erel}} e'_1$ (so $\underline{L'_R \vdash_{\text{erel}} e'_1[\tau']}$), and $C' \vdash_{\text{rtyp}} e'_1 : \forall\alpha{:}\kappa[\gamma].\tau''$. By the Context Weakening Lemma $L'; \cdot \vdash_{\text{k}} \tau' : \kappa$. So with $\cdot \vdash_{\text{eff}} \gamma$ we can derive $\underline{C' \vdash_{\text{rtyp}} e'_1[\tau'] : \tau''[\tau'/\alpha]}$.

  If $e = (e_1, e_2)$ and e_1 is not a value, let $e' = (e'_1, e_2)$. Inverting $\vdash_{\text{j}} H_j; (e_1, e_2)$ ensures $\vdash_{\text{j}} H_j; e_1$ and $\vdash_{\text{jf}} e_2$. Inverting $L_R \vdash_{\text{erel}} (e_1, e_2)$ ensures $L_R \vdash_{\text{erel}} e_1$ and $\cdot \vdash_{\text{erel}} e_2$. Inverting $C \vdash_{\text{rtyp}} (e_1, e_2) : \tau$ ensures $\tau = \tau_1 \times \tau_2$, $C \vdash_{\text{rtyp}} e_1 : \tau_1$, and $C \vdash_{\text{rtyp}} e_2 : \tau_2$. So the induction hypothesis provides $\vdash_{\text{j}} H'_j; e'_1$ (so $\underline{\vdash_{\text{j}} H'_j; (e'_1, e_2)}$), $L'_R \vdash_{\text{erel}} e'_1$ (so $\underline{L'_R \vdash_{\text{erel}} (e'_1, e_2)}$), and $C' \vdash_{\text{rtyp}} e'_1 : \tau_1$. By the Term Weakening Lemma $C' \vdash_{\text{rtyp}} e_2 : \tau_2$, so we can derive $\underline{C' \vdash_{\text{rtyp}} (e'_1, e_2) : \tau_1 \times \tau_2}$.

  If $e = (v, e_1)$, let $e' = (v, e'_1)$. Inverting $\vdash_{\text{j}} H_j; (v, e_1)$ ensures $\vdash_{\text{jf}} v$ and $\vdash_{\text{j}} H_j; e_1$ (because the Values Effectless Lemma ensures $x, v' \not\vdash_{\text{e}} v$). Inverting $L_R \vdash_{\text{erel}} (v, e_1)$ ensures $\cdot \vdash_{\text{erel}} v$ and $L_R \vdash_{\text{erel}} e_1$ (because if $L_R \vdash_{\text{erel}} v$, then $\cdot \vdash_{\text{erel}} e_1$ and

the Values Effectless Lemma ensures $L_R = \cdot$). Inverting $C \vdash_{\text{rtyp}} (v, e_1) : \tau$ ensures $\tau = \tau_0 \times \tau_1$, $C \vdash_{\text{rtyp}} v : \tau_0$, and $C \vdash_{\text{rtyp}} e_1 : \tau_1$. So the induction hypothesis provides $\vdash_{\text{j}} H'_j; e'_1$ (so $\underline{\vdash_{\text{j}} H'_j; (v, e'_1)}$), $L'_R \vdash_{\text{erel}} e'_1$ (so $\underline{L'_R \vdash_{\text{erel}} (v, e'_1)}$), and $C' \vdash_{\text{rtyp}} e'_1 : \tau_1$. By the Term Weakening Lemma $C' \vdash_{\text{rtyp}} v : \tau_0$, so we can derive $\underline{C' \vdash_{\text{rtyp}} (v, e'_1) : \tau_0 \times \tau_1}$.

If $e = e_1(e_2)$ and e_1 is not a value, let $e' = e'_1(e_2)$. Inverting $\vdash_{\text{j}} H_j; e_1(e_2)$ ensures $\vdash_{\text{j}} H_j; e_1$ and $\vdash_{\text{jf}} e_2$. Inverting $L_R \vdash_{\text{erel}} e_1(e_2)$ ensures $L_R \vdash_{\text{erel}} e_1$ and $\cdot \vdash_{\text{erel}} e_2$. Inverting $C \vdash_{\text{rtyp}} e_1(e_2) : \tau$ ensures $C \vdash_{\text{rtyp}} e_1 : \tau_1 \xrightarrow{\epsilon_1} \tau$, $C \vdash_{\text{rtyp}} e_2 : \tau_1$, and $\cdot \vdash_{\text{eff}} \epsilon_1 \subseteq \epsilon$. So the induction hypothesis provides $\vdash_{\text{j}} H'_j; e'_1$ (so $\underline{\vdash_{\text{j}} H'_j; e'_1(e_2)}$), $L'_R \vdash_{\text{erel}} e'_1$ (so $\underline{L'_R \vdash_{\text{erel}} e'_1(e_2)}$), and $C' \vdash_{\text{rtyp}} e'_1 : \tau_1 \xrightarrow{\epsilon_1} \tau$. By the Term Weakening Lemma $C' \vdash_{\text{rtyp}} e_2 : \tau_1$. So because $C'_\gamma = \cdot$ and $C'_\epsilon = \epsilon$, we can derive $\underline{C' \vdash_{\text{rtyp}} e'_1(e_2) : \tau}$.

If $e = v(e_1)$, let $e' = v(e'_1)$. Inverting $\vdash_{\text{j}} H_j; v(e_1)$ ensures $\vdash_{\text{jf}} v$ and $\vdash_{\text{j}} H_j; e_1$ (because the Values Effectless Lemma ensures $x, v' \not\vdash_{\text{e}} v$). Inverting $L_R \vdash_{\text{erel}} v(e_1)$ ensures $\cdot \vdash_{\text{erel}} v$ and $L_R \vdash_{\text{erel}} e_1$ (because if $L_R \vdash_{\text{erel}} v$, then $\cdot \vdash_{\text{erel}} e_1$ and the Values Effectless Lemma ensures $L_R = \cdot$). Inverting $C \vdash_{\text{rtyp}} v(e_1) : \tau$ ensures $C \vdash_{\text{rtyp}} v : \tau_1 \xrightarrow{\epsilon_1} \tau$, $C \vdash_{\text{rtyp}} e_1 : \tau_1$, and $\cdot \vdash_{\text{eff}} \epsilon_1 \subseteq \epsilon$. So the induction hypothesis provides $\vdash_{\text{j}} H'_j; e'_1$ (so $\underline{\vdash_{\text{j}} H'_j; v(e'_1)}$), $L'_R \vdash_{\text{erel}} e'_1$ (so $\underline{L'_R \vdash_{\text{erel}} v(e'_1)}$), and $C' \vdash_{\text{rtyp}} e'_1 : \tau_1$. By the Term Weakening Lemma $C' \vdash_{\text{rtyp}} v : \tau_1 \xrightarrow{\epsilon_1} \tau$. So because $C'_\gamma = \cdot$ and $C'_\epsilon = \epsilon$, we can derive $\underline{C' \vdash_{\text{rtyp}} v(e'_1) : \tau}$.

If $e = \mathsf{pack}\ \tau_1, e_1\ \mathsf{as}\ \tau_2$, let $e' = \mathsf{pack}\ \tau_1, e'_1\ \mathsf{as}\ \tau_2$. Inverting $\vdash_{\text{j}} H_j; \mathsf{pack}\ \tau_1, e_1\ \mathsf{as}\ \tau_2$ ensures $\vdash_{\text{j}} H_j; e_1$. Inverting $L_R \vdash_{\text{erel}} \mathsf{pack}\ \tau_1, e_1\ \mathsf{as}\ \tau_2$ ensures $L_R \vdash_{\text{erel}} e_1$. Inverting $C \vdash_{\text{rtyp}} \mathsf{pack}\ \tau_1, e_1\ \mathsf{as}\ \tau_2 : \tau$ ensures $\tau_2 = \tau = \exists\alpha{:}\kappa[\gamma].\tau_3$, $C \vdash_{\text{rtyp}} e_1 : \tau_3[\tau_1/\alpha]$, $L; \cdot \vdash_{\text{k}} \tau_1 : \kappa$, $\cdot \vdash_{\text{eff}} \gamma[\tau_1/\alpha]$, and $L; \cdot \vdash_{\text{k}} \exists\alpha{:}\kappa[\gamma].\tau_3 : \mathtt{AU}$. So the induction hypothesis provides $\vdash_{\text{j}} H_j; e'_1$ (so $\underline{\vdash_{\text{j}} H'_j; \mathsf{pack}\ \tau_1, e'_1\ \mathsf{as}\ \tau_2}$), $L'_R \vdash_{\text{erel}} e'_1$ (so $\underline{L'_R \vdash_{\text{erel}} \mathsf{pack}\ \tau_1, e'_1\ \mathsf{as}\ \tau_2}$), and $C' \vdash_{\text{rtyp}} e_1 : \tau_3[\tau_1/\alpha]$. The Context Weakening Lemma ensures $L'; \cdot \vdash_{\text{k}} \tau_1 : \kappa$ and $L'; \cdot \vdash_{\text{k}} \exists\alpha{:}\kappa[\gamma].\tau_3 : \mathtt{AU}$. So because $C'_\gamma = \cdot$ and $\cdot \vdash_{\text{eff}} \gamma[\tau_1/\alpha]$, we can derive $\underline{C' \vdash_{\text{rtyp}} \mathsf{pack}\ \tau_1, e'_1\ \mathsf{as}\ \exists\alpha{:}\kappa[\gamma].\tau_3 : \exists\alpha{:}\kappa[\gamma].\tau_3}$.

- DS5.1: Let $s = \mathsf{let}\ \ell, x{=}v;\ s_1$ and $s' = s_1$. Inverting $C; \tau \vdash_{\text{styp}} \mathsf{let}\ \ell, x{=}v;\ s_1$ ensures $C \vdash_{\text{rtyp}} v : \tau_1$ and $L; \cdot; \Gamma_S \Gamma_U, x{:}(\tau_1, \ell); \cdot; \epsilon; \tau \vdash_{\text{styp}} s_1$. So the Typing Well-Formedness Lemma and inversion ensures $L; \cdot \vdash_{\text{k}} \ell : \mathtt{LU}$, so the Type Canonical Forms Lemma ensures $\ell = \mathit{loc}$ or $\ell = \mathrm{S}(i)$ for some $i \in L$ (so i is in one of L_X, L_0, L_R, or L_E). Our choice of Γ'_S, Γ'_U, H'_{XS}, $H_{0S'}$, H'_S, and H'_U depends on ℓ and τ_1:

  1. If $\ell = \mathit{loc}$ or $L; \cdot \not\vdash_{\text{k}} \tau_1 : \mathtt{AS}$, let $H'_{XS} = H_{XS}$, $H'_{0S} = H_{0S}$, $H'_S = H_S$, and $H'_U = H_U, x \mapsto v$. Let $\Gamma'_S = \Gamma_S$ and $\Gamma'_U = \Gamma_U, , x{:}(\tau_1, \ell)$.

2. Else if $\ell = \mathrm{S}(i)$ for some $i \in L_R L_E$, let $H'_{XS} = H_{XS}$, $H'_{0S} = H_{0S}$, $H'_S = H_S, x \mapsto v$, and $H'_U = H_U$. Let $\Gamma'_S = \Gamma_S, x{:}(\tau_1, \ell)$ and $\Gamma'_U = \Gamma_U$.
3. Else if $\ell = \mathrm{S}(i)$ and $i \in L_X$, let $H'_{XS} = H_{XS}, x \mapsto v$, $H'_{0S} = H_{0S}$, $H'_S = H_S$, and $H'_U = H_U$. Let $\Gamma'_S = \Gamma_S, x{:}(\tau_1, \ell)$ and $\Gamma_U = \Gamma_U$.
4. Else if $\ell = \mathrm{S}(i)$ and $i \in L_0$, let $H'_{XS} = H_{XS}$, $H'_{0S} = H_{0S}, x \mapsto v$, $H'_S = H_S$, and $H'_U = H_U$. Let $\Gamma'_S = \Gamma_S, x{:}(\tau_1, \ell)$ and $\Gamma'_U = \Gamma_U$.

In all cases, letting $L'_R = L_R$ and $L'_0 = L_0$, all conclusions except Conclusion 3 follow easily. For 3, we first show $\vdash_{\text{hind}} H'_{XS}; H'_{0S}; H'_S; H'_U; L; L_0; L_X; L_R; L_E : \Gamma'_S; \Gamma'_U; \epsilon$ given the $\vdash_{\text{hind}}$ assumption, proceeding by cases:

1. All obligations are immediate except $L; \Gamma_S \Gamma_U, x{:}(\tau_1, \ell) \vdash_{\text{htyp}} H_U, x \mapsto v : \Gamma_U, x{:}(\tau_1, \ell)$ (which follows from $L; \Gamma_S \Gamma_U \vdash_{\text{htyp}} H_U : \Gamma_U$, the Heap Weakening Lemma, and $C \vdash_{\text{rtyp}} v : \tau_1$) and $L \vdash_{\text{loc}} \Gamma_U, x{:}(\tau_1, \ell)$ (which follows from $L \vdash_{\text{loc}} \Gamma_U$ and $L; \cdot \not\vdash_{\text{k}} \tau_1 : \mathtt{AS}$).
2. All obligations are, with possible use of the Heap Weakening Lemma, immediate except $\Gamma'_S; L_R L_E \vdash_{\text{hlk}} H_S, x \mapsto v$ (which follows from $\Gamma_S; L_R L_E \vdash_{\text{hlk}} H_S$, the Heap Weakening Lemma, $\Gamma'_S(x) = (\tau_1, \mathrm{S}(i))$, and $i \in L_R L_E$), $L; \Gamma'_S \vdash_{\text{htyp}} H_{XS} H_{0S} H_S, x \mapsto v : \Gamma'_S$ (which follows from $L; \Gamma_S \vdash_{\text{htyp}} H_{XS} H_{0S} H_S : \Gamma_S$, the Heap Weakening Lemma, $C \vdash_{\text{rtyp}} v : \tau_1$, $L; \cdot \vdash_{\text{k}} \tau_1 : \mathtt{AS}$, and the Sharable Values Need Only Sharable Context Lemma), and $L \vdash_{\text{shr}} \Gamma'_S$ (which follows from $L \vdash_{\text{shr}} \Gamma_S$, $L; \cdot \vdash_{\text{k}} \tau_1 : \mathtt{AS}$, and the directly derivable $L; \cdot \vdash_{\text{k}} \mathrm{S}(i) : \mathtt{LS}$).
3. This case is the same as case 2 except we use $\Gamma_S; L_X \vdash_{\text{hlk}} H_{XS}$ and $i \in L_X$ to show $\Gamma'_S; L_X \vdash_{\text{hlk}} H_{XS}, x \mapsto v$.
4. This case is the same as case 2 except we use $\Gamma_S; L_0 \vdash_{\text{hlk}} H_{0S}$ and $i \in L_0$ to show $\Gamma'_S; L_0 \vdash_{\text{hlk}} H_{0S}, x \mapsto v$ and we must show $\vdash_{\text{jf}} H_{0S}, x \mapsto v$. The latter follows from $\vdash_{\text{jf}} H_{0S}$ and $\vdash_{\text{jf}} v$ (which we prove below).

Only the other $\vdash_{\text{sind}}$ obligations remain. From $L; \cdot; \Gamma_S \Gamma_U, x{:}(\tau_1, \ell); \cdot; \epsilon; \tau \vdash_{\text{styp}} s_1$ and reordering, we have $\underline{L; \cdot; \Gamma'_S \Gamma'_U; \cdot; \epsilon; \tau \vdash_{\text{styp}} s_1}$. The Values Effectless Lemma and inverting $L_R \vdash_{\text{srel}} \mathsf{let}\ \ell, x{=}v;\ s_1$ ensure $L_R = \cdot$ and $\cdot \vdash_{\text{srel}} s_1$, i.e, $\underline{L_R \vdash_{\text{srel}} s_1}$. The Values Effectless Lemma and inverting $\vdash_{\text{j}} H_j; \mathsf{let}\ \ell, x{=}v;\ s_1$ ensure $\vdash_{\text{jf}} H_j$, $\vdash_{\text{jf}} v$, and $\vdash_{\text{jf}} s_1$, so in each of the four cases $\vdash_{\text{jf}} H'_j$ and therefore $\underline{\vdash_{\text{j}} H'_j; s_1}$.

- DS5.2: Let $s = (v; s_1)$ and $s' = s_1$. This case is local. By the Values Effectless Lemma, $x, v' \not\vdash_{\text{e}} v$, so inverting $\vdash_{\text{j}} H_j; (v; s_1)$ ensures $\vdash_{\text{jf}} H_j$ and $\vdash_{\text{jf}} s_1$. So $\underline{\vdash_{\text{j}} H_j; s_1}$. The Values Effectless Lemma and inverting $L_R \vdash_{\text{srel}} v; s_1$

ensures $L_R = \cdot$ and $\cdot \vdash_{\text{srel}} s_1$, i.e., $\underline{L_R \vdash_{\text{srel}} s_1}$. Inverting $C; \tau \vdash_{\text{styp}} v; s_1$ provides $\underline{C; \tau \vdash_{\text{styp}} s_1}$.

- DS5.3: Let $s = (\mathsf{return}\ v; s_1)$ and $s' = \mathsf{return}\ v$. This case is local. Inverting $\vdash_{\text{j}} H_j; (\mathsf{return}\ v; s_1)$ ensures $\underline{\vdash_{\text{j}} H_j; \mathsf{return}\ v}$. Inverting $L_R \vdash_{\text{erel}} \mathsf{return}\ v; s_1$ ensures $L_R \vdash_{\text{erel}} \mathsf{return}\ v$. Inverting $C; \tau \vdash_{\text{styp}} \mathsf{return}\ v; s_1$ ensures $C; \tau \vdash_{\text{styp}} \mathsf{return}\ v$.

- DS5.4: Let $s = \mathsf{if}\ 0\ s_1\ s_2$ and $s' = s_2$. This case is local. Because $x, v \not\vdash_{\text{e}} 0$, inverting $\vdash_{\text{j}} H_j; \mathsf{if}\ 0\ s_1\ s_2$ ensures $\vdash_{\text{jf}} H_j$ and $\vdash_{\text{jf}} s_2$. So $\underline{\vdash_{\text{j}} H_j; s_2}$. Because $L_R \vdash_{\text{erel}} 0$ ensures $L_R = \cdot$, inverting $L_R \vdash_{\text{erel}} \mathsf{if}\ 0\ s_1\ s_2$ ensures $\cdot \vdash_{\text{erel}} s_2$, i.e., $\underline{L_R \vdash_{\text{srel}} s_2}$. Inverting $C; \tau \vdash_{\text{styp}} \mathsf{if}\ 0\ s_1\ s_2$ ensures $\underline{C; \tau \vdash_{\text{styp}} s_2}$.

- DS5.5: This case is analogous to the previous one.

- DS5.6: Let $s = \mathsf{while}\ e\ s_1$ and $s' = \mathsf{if}\ e\ (s_1; \mathsf{while}\ e\ s_1)\ 0$. This case is local. Inverting $\vdash_{\text{j}} H_j; \mathsf{while}\ e\ s_1$ ensures $\vdash_{\text{jf}} H_j$, $\vdash_{\text{jf}} e$, and $\vdash_{\text{jf}} s_1$. So because $\vdash_{\text{jf}} 0$, we can derive $\underline{\vdash_{\text{j}} H_j; \mathsf{if}\ e\ (s_1; \mathsf{while}\ e\ s_1)\ 0}$. Inverting $L_R \vdash_{\text{srel}} \mathsf{while}\ e\ s_1$ ensures $L_R = \cdot$, $\cdot \vdash_{\text{erel}} e$, and $\cdot \vdash_{\text{srel}} s_1$. So because $\cdot \vdash_{\text{erel}} 0$, we can derive $\underline{L_R \vdash_{\text{srel}} \mathsf{if}\ e\ (s_1; \mathsf{while}\ e\ s_1)\ 0}$. Inverting $C; \tau \vdash_{\text{styp}} \mathsf{while}\ e\ s_1$ ensures $C \vdash_{\text{rtyp}} e : \text{int}$ and $C; \tau \vdash_{\text{styp}} s_1$. So because $C \vdash_{\text{rtyp}} 0 : \text{int}$, we can derive $\underline{C; \tau \vdash_{\text{styp}} \mathsf{if}\ e\ (s_1; \mathsf{while}\ e\ s_1)\ 0}$.

- DS5.7: Let $s = \mathsf{open}\ (\mathsf{pack}\ \tau_1, v\ \mathsf{as}\ \exists\alpha{:}\kappa[\gamma].\tau_2)\ \mathsf{as}\ \ell, \alpha, x;\ s_1$ and $s' = (\mathsf{let}\ \ell, x{=}v;\ s_1[\tau_1/\alpha])$. This case is local. Inverting $\vdash_{\text{j}} H_j; s$ ensures $\vdash_{\text{j}} H_j; v$ and $\vdash_{\text{jf}} s_1$, so Term Substitution Lemma 3 ensures $\underline{\vdash_{\text{j}} H_j; \mathsf{let}\ \ell, x{=}v;\ s_1[\tau_1/\alpha]}$. Inverting $L_R \vdash_{\text{srel}} s$ ensures $L_R \vdash_{\text{erel}} v$ and $\cdot \vdash_{\text{srel}} s_1$, so Term Substitution Lemma 2 ensures $\underline{L_R \vdash_{\text{srel}} \mathsf{let}\ \ell, x{=}v;\ s_1[\tau_1/\alpha]}$. Inverting $C; \tau \vdash_{\text{styp}} s$ ensures $C \vdash_{\text{rtyp}} v : \tau_2[\tau_1/\alpha]$, $\cdot \vdash_{\text{eff}} \gamma[\tau_1/\alpha]$, $L; \cdot \vdash_{\text{k}} \tau_1 : \kappa$, $L; \alpha{:}\kappa; \Gamma_S\Gamma_U, x{:}(\tau_2, \ell); \gamma; \epsilon; \tau \vdash_{\text{styp}} s_1$, $L; \cdot \vdash_{\text{k}} \ell : \mathtt{LU}$, and $L; \cdot \vdash_{\text{k}} \tau : \mathtt{AU}$. So Term Substitution Lemma 4 ensures $L; \cdot; (\Gamma_S\Gamma_U, x{:}(\tau_2, \ell))[\tau_1/\alpha]; \gamma[\tau_1/\alpha]; \epsilon[\tau_1/\alpha]; \tau[\tau_1/\alpha] \vdash_{\text{styp}} s_1[\tau_1/\alpha]$. The Typing Well-Formedness Lemma ensures $\vdash_{\text{wf}} C$, so the Useless Substitution Lemma and the kinding for ℓ and τ ensure $L; \cdot; \Gamma_S\Gamma_U, x{:}(\tau_2[\tau_1/\alpha], \ell); \gamma[\tau_1/\alpha]; \epsilon; \tau \vdash_{\text{styp}} s_1[\tau_1/\alpha]$. Because $\cdot \vdash_{\text{eff}} \gamma[\tau_1/\alpha]$, the Term Weakening Lemma ensures $L; \cdot; \Gamma_S\Gamma_U, x{:}(\tau_2[\tau_1/\alpha], \ell); \cdot; \epsilon; \tau \vdash_{\text{styp}} s_1[\tau_1/\alpha]$. So with $C \vdash_{\text{rtyp}} v : \tau_2[\tau_1/\alpha]$, we can derive $\underline{L; \cdot; \Gamma_S\Gamma_U; \cdot; \epsilon; \tau \vdash_{\text{styp}} \mathsf{let}\ \ell, x{=}v;\ s_1[\tau_1/\alpha]}$.

- DS5.8: Let $s = \mathsf{sync}\ \mathsf{lock}\ i\ s_1$ and $s' = s_1; \mathsf{release}\ i$. Also let $L_0 = L_0', i$. The $\vdash_{\text{hind}}$ assumption ensures $\Gamma_S; L_0 \vdash_{\text{hlk}} H_{0S}$. A trivial induction on this derivation ensures we can write H_{0S} as $H_{0S}'H_i$ such that $\Gamma_S; L_0' \vdash_{\text{hlk}} H_{0S}'$ and $\Gamma_S; i \vdash_{\text{hlk}} H_i$. Inverting $L_R \vdash_{\text{srel}} \mathsf{sync}\ \mathsf{lock}\ i\ s_1$ ensures $L_R = \cdot$ and $\cdot \vdash_{\text{srel}} s_1$. Letting $H_{XS}' = H_{XS}$, $H_S' = H_SH_i$, $H_U' = H_U$, $\Gamma_S' = \Gamma_S$, $\Gamma_U' = \Gamma_U$, and

$L'_R = i$, all of the conclusions follow immediately except for Conclusion 3. (Note that $H' = H$ and $L' = L$.)

First we show $\vdash_{\text{hind}} H_{XS}; H'_{0S}; H'_S; H_U; L; L'_0; L_X; i; L_E : \Gamma_S; \Gamma_U; \epsilon$ given $\vdash_{\text{hind}}$. We know $\underline{\vdash_{\text{jf}} H'_{0S}}$ because $\vdash_{\text{jf}} H_{0S}$ and $H_{0S} = H'_{0S}H_i$. We argued above that $\underline{\Gamma_S; L'_0 \vdash_{\text{hlk}} H'_{0S}}$. Because $\Gamma_S; \cdot L_E \vdash_{\text{hlk}} H_S$ and $\Gamma_S; i \vdash_{\text{hlk}} H_i$, a trivial induction shows $\underline{\Gamma_S; iL_E \vdash_{\text{hlk}} H'_S}$. All other obligations are provided directly from the $\vdash_{\text{hind}}$ assumption because $H_{XS}H'_{0S}H'_S = H_{XS}H_{0S}H_S$ and $L'_0L_XiL_E = L_0L_XL_RL_E$.

To conclude $\vdash_{\text{sind}} H_{XS}; H'_{0S}; H'_S; H_U; L; L'_0; L_X; i; L_E; \tau; s_1; \mathsf{release}\ i : \Gamma_S; \Gamma_U$, we still must show $\vdash_{\text{j}} H_SH_iH_U; s_1; \mathsf{release}\ i$, $C; \tau \vdash_{\text{styp}} s_1; \mathsf{release}\ i$, and $i \vdash_{\text{srel}} s_1; \mathsf{release}\ i$. Inverting $\vdash_{\text{j}} H_j; \mathsf{sync\ lock}\ i\ s_1$ ensures $\vdash_{\text{jf}} H_S$, $\vdash_{\text{jf}} H_U$, and $\vdash_{\text{jf}} s_1$. From the $\vdash_{\text{hind}}$ assumption, we know $\vdash_{\text{jf}} H_{0S}$ so $\vdash_{\text{jf}} H_i$. So we can derive $\underline{\vdash_{\text{j}} H_SH_iH_U; s_1; \mathsf{release}\ i}$. We showed above that $\cdot \vdash_{\text{srel}} s_1$, so we can derive $\underline{i \vdash_{\text{srel}} s_1; \mathsf{release}\ i}$. Inverting $C; \tau \vdash_{\text{styp}} \mathsf{sync\ lock}\ i\ s_1$ ensures $C \vdash_{\text{rtyp}} \mathsf{lock}\ i : \text{lock}(\text{S}(i))$ and $L; \cdot; \Gamma_S\Gamma_U; \cdot; \epsilon \cup \text{locks}(\text{S}(i)); \tau \vdash_{\text{styp}} s_1$. Because $\text{locks}(\text{S}(i)) = i$, we can derive $\underline{C; \tau \vdash_{\text{styp}} s_1; \mathsf{release}\ i}$.

- DS5.9: Let $s = \mathsf{sync\ nonlock}\ s_1$ and $s' = s_1$. This case is local. Because $x, v \not\in_{\text{e}} \mathsf{nonlock}$, inverting $\vdash_{\text{j}} H_j; \mathsf{sync\ nonlock}\ s_1$ ensures $\vdash_{\text{jf}} H_j$ and $\vdash_{\text{jf}} s_1$. So $\underline{\vdash_{\text{j}} H_j; s_1}$. Because $L_R \vdash_{\text{erel}} \mathsf{nonlock}$ ensures $L_R = \cdot$, inverting $L_R \vdash_{\text{srel}} \mathsf{sync\ nonlock}\ s_1$ ensures $L_R = \cdot$ and $\cdot \vdash_{\text{srel}} s_1$, i.e., $\underline{L_R \vdash_{\text{srel}} s_1}$. Because $\text{locks}(\mathsf{nonlock}) = \emptyset$, inverting $C; \tau \vdash_{\text{styp}} \mathsf{sync\ nonlock}\ s_1$ ensures $\underline{C; \tau \vdash_{\text{styp}} s_1}$.

- DS5.10: Let $s = v; \mathsf{release}\ i$ and $s' = v$. The Values Effectless Lemma and inverting $L_R \vdash_{\text{srel}} v; \mathsf{release}\ i$ ensures $L_R = i$ and $\cdot \vdash_{\text{srel}} v$. The $\vdash_{\text{hind}}$ assumption ensures $\Gamma_S; L_RL_E \vdash_{\text{hlk}} H_S$. A trivial induction on this derivation ensures we can write H_S as H'_SH_i such that $\Gamma_S; L_E \vdash_{\text{hlk}} H'_S$ and $\Gamma_S; i \vdash_{\text{hlk}} H_i$. Letting $H'_{XS} = H_{XS}$, $H'_{0S} = H_{0S}H_i$, $H'_U = H_U$, $\Gamma'_S = \Gamma_S$, $\Gamma'_U = \Gamma_U$, $L'_0 = L_0, i$, and $L'_R = \cdot$, all of the conclusions follow immediately except for Conclusion 3. (Note that $H' = H$ and $L' = L$.)

  First we show $\vdash_{\text{hind}} H_{XS}; H'_{0S}; H'_S; H_U; L; L'_0; L_X; \cdot; L_E : \Gamma_S; \Gamma_U; \epsilon$ given $\vdash_{\text{hind}}$. The Values Effectless Lemma and inverting $\vdash_{\text{j}} H_j; v; \mathsf{release}\ i$ ensures $\vdash_{\text{jf}} v$ and $\vdash_{\text{jf}} H_j$, so therefore $\vdash_{\text{jf}} H_i$. So with $\vdash_{\text{jf}} H_{0S}$, we know $\underline{\vdash_{\text{jf}} H'_{0S}}$. Because $\Gamma_S; L_0 \vdash_{\text{hlk}} H_{0S}$ and $\Gamma_S; i \vdash_{\text{hlk}} H_i$, a trivial induction shows $\underline{\Gamma_S; L'_0 \vdash_{\text{hlk}} H'_{0S}}$. We argued above that $\underline{\Gamma_S; L_E \vdash_{\text{hlk}} H'_S}$. All other obligations are provided directly from the $\vdash_{\text{hind}}$ assumption because $H_{XS}H'_{0S}H'_S = H_{XS}H_{0S}H_S$ and $L'_0L_XiL_E = L_0L_XL_RL_E$.

  To conclude $\vdash_{\text{sind}} H_{XS}; H'_{0S}; H'_S; H_U; L; L'_0; L_X; \cdot; L_E; \tau; v : \Gamma_S; \Gamma_U$, we still must show $\vdash_{\text{j}} H'_SH_U; v$, $C; \tau \vdash_{\text{styp}} v$, and $\cdot \vdash_{\text{srel}} v$. We already showed

$\vdash_{\text{jf}} H_j$ (so $\vdash_{\text{jf}} H'_j$) and $\vdash_{\text{jf}} v$, so $\vdash_{\text{j}} H'_j; v$. Inverting $C; \tau \vdash_{\text{styp}} v; \mathsf{release}\ i$ ensures $L; \cdot; \Gamma_S \Gamma_U; \cdot; \epsilon \cup i; \tau \vdash_{\text{styp}} v$, so the Values Effectless Lemma ensures $C; \tau \vdash_{\text{styp}} v$. We already showed $\cdot \vdash_{\text{srel}} v$.

- DS5.11: Let $s = \mathsf{return}\ v; \mathsf{release}\ i$. This case is analogous to the previous one because inversion ensures $C \vdash_{\text{rtyp}} v : \tau$, $\cdot \vdash_{\text{erel}} v$, and $\vdash_{\text{j}} H_j; v$.

- DS5.12: Let $s = \mathsf{spawn}\ v_1(v_2)$, $s' = 0$, and $s_{\text{opt}} = \mathsf{return}\ v_1 v_2$. Leaving all heap portions and lock sets unchanged ($H'_{XS} = H_{XS}$, $H'_{0S} = H_{0S}$, $H'_S = H_S$, $H'_U = H_U$, $\Gamma'_S = \Gamma_S$, $\Gamma'_U = \Gamma_U$, $L'_0 = L_0$, and $L'_R = L_R$), all conclusions except 3 and 7 are trivial. For 3, the Values Effectless Lemma and inversion of $L_R \vdash_{\text{srel}} \mathsf{spawn}\ v_1(v_2)$ ensures $L_R = \cdot$, $\cdot \vdash_{\text{erel}} v_1$, and $\cdot \vdash_{\text{erel}} v_2$, so $L_R \vdash_{\text{srel}} 0$. The Values Effectless Lemma and inversion of $\vdash_{\text{jf}} H_j; \mathsf{spawn}\ v_1(v_2)$ ensures $\vdash_{\text{jf}} H_j$, $\vdash_{\text{jf}} v_1$, and $\vdash_{\text{jf}} v_2$. So $\vdash_{\text{j}} H_j; 0$. The Typing Well-Formedness Lemma ensures $C; \tau \vdash_{\text{styp}} 0$. With the $\vdash_{\text{hind}}$ assumption, the underlined facts establish Conclusion 3.

  For 7, $\vdash_{\text{ret}} \mathsf{return}\ v_1(v_2)$ is trivial. We showed above $\cdot \vdash_{\text{erel}} v_1$ and $\cdot \vdash_{\text{erel}} v_2$, so $\cdot \vdash_{\text{srel}} \mathsf{return}\ v_1(v_2)$. We showed above $\vdash_{\text{jf}} v_1$ and $\vdash_{\text{jf}} v_2$, so $\vdash_{\text{jf}} \mathsf{return}\ v_1(v_2)$. Inverting $C; \tau \vdash_{\text{styp}} \mathsf{spawn}\ v_1(v_2)$ ensures $C \vdash_{\text{rtyp}} v_1 : \tau_1 \xrightarrow{\emptyset} \tau_2$, $C \vdash_{\text{rtyp}} v_2 : \tau_1$, and $L; \cdot \vdash_{\text{k}} \tau_1 : \mathtt{AS}$. The Typing Well-Formedness Lemma ensures $L; \cdot \vdash_{\text{k}} \tau_1 \xrightarrow{\emptyset} \tau_2 : \mathtt{AU}$, so the Type Canonical Forms Lemma ensures $L; \cdot \vdash_{\text{k}} \tau_1 \xrightarrow{\emptyset} \tau_2 : \mathtt{AS}$. So two uses of the Sharable Values Need Only Sharable Context Lemma ensure $L; \cdot; \Gamma_S; \cdot; \epsilon \vdash_{\text{rtyp}} v_1 : \tau_1 \xrightarrow{\emptyset} \tau_2$ and $L; \cdot; \Gamma_S; \cdot; \epsilon \vdash_{\text{rtyp}} v_2 : \tau_1$. So the Values Effectless Lemma ensures $L; \cdot; \Gamma_S; \cdot; \emptyset \vdash_{\text{rtyp}} v_1 : \tau_1 \xrightarrow{\emptyset} \tau_2$ and $L; \cdot; \Gamma_S; \cdot; \emptyset \vdash_{\text{rtyp}} v_2 : \tau_1$. So we can derive $L; \cdot; \Gamma_S; \cdot; \emptyset; \tau_2 \vdash_{\text{styp}} \mathsf{return}\ v_1(v_2)$.

- DS5.13: There are eight inductive cases. If $s = e$, let $s' = e'$. Inverting $\vdash_{\text{j}} H_j; s$ ensures $\vdash_{\text{j}} H_j; e$. Inverting $L_R \vdash_{\text{srel}} e$ ensures $L_R \vdash_{\text{erel}} e$. Inverting $C; \tau \vdash_{\text{styp}} e$ ensures $C \vdash_{\text{rtyp}} e : \tau'$. So the induction hypothesis provides $\vdash_{\text{j}} H'_j; e'$ (so $\vdash_{\text{j}} H'_j; s'$), $L'_R \vdash_{\text{erel}} e'$ (so $L'_R \vdash_{\text{srel}} s'$), and $C' \vdash_{\text{rtyp}} e' : \tau'$ (so $C'; \tau \vdash_{\text{styp}} s'$).

  If $s = \mathsf{return}\ e$, the argument is analogous to the case $s = e$. Note that $\tau' = \tau$.

  If $s = \mathsf{if}\ e\ s_1\ s_2$, let $s' = \mathsf{if}\ e'\ s_1\ s_2$. Inverting $\vdash_{\text{j}} H_j; \mathsf{if}\ e\ s_1\ s_2$ ensures $\vdash_{\text{j}} H_j; e$, $\vdash_{\text{jf}} s_1$, and $\vdash_{\text{jf}} s_2$. Inverting $L_R \vdash_{\text{srel}} \mathsf{if}\ e\ s_1\ s_2$ ensures $L_R \vdash_{\text{erel}} e$, $\cdot \vdash_{\text{srel}} s_1$, and $\cdot \vdash_{\text{srel}} s_2$. Inverting $C; \tau \vdash_{\text{styp}} \mathsf{if}\ e\ s_1\ s_2$ ensures $C \vdash_{\text{rtyp}} e : \text{int}$, $C; \tau \vdash_{\text{styp}} s_1$, and $C; \tau \vdash_{\text{styp}} s_2$. So the induction hypothesis provides $\vdash_{\text{j}} H'_j; e'$ (so $\vdash_{\text{j}} H'_j; \mathsf{if}\ e'\ s_1\ s_2$), $L'_R \vdash_{\text{erel}} e'$ (so $L'_R \vdash_{\text{srel}} \mathsf{if}\ e'\ s_1\ s_2$), and $C' \vdash_{\text{rtyp}} e' : \text{int}$. The Term Weakening Lemma ensures $C'; \tau \vdash_{\text{styp}} s_1$ and $C'; \tau \vdash_{\text{styp}} s_2$, so $C'; \tau \vdash_{\text{styp}} \mathsf{if}\ e'\ s_1\ s_2$.

If $s = \mathsf{let}\ \ell, x{=}e;\ s_1$, let $s' = \mathsf{let}\ \ell, x{=}e';\ s_1$. Inverting $\vdash_{\mathtt{j}} H_j; \mathsf{let}\ \ell, x{=}e;\ s_1$ ensures $\vdash_{\mathtt{j}} H_j; e$ and $\vdash_{\mathtt{jf}} s_1$. Inverting $L_R \vdash_{\mathtt{srel}} \mathsf{let}\ \ell, x{=}e;\ s_1$ ensures $L_R \vdash_{\mathtt{erel}} e$ and $\cdot \vdash_{\mathtt{srel}} s_1$. Inverting $C; \tau \vdash_{\mathtt{styp}} \mathsf{let}\ \ell, x{=}e;\ s_1$ ensures $C \vdash_{\mathtt{rtyp}} e : \tau'$ and $L; \cdot; \Gamma_S \Gamma_U, x{:}(\tau', \ell); \cdot; \epsilon; \tau \vdash_{\mathtt{styp}} s_1$. So the induction hypothesis provides $\vdash_{\mathtt{j}} H'_j; e'$ (so $\underline{\vdash_{\mathtt{j}} H'_j; \mathsf{let}\ \ell, x{=}e';\ s_1}$), $L'_R \vdash_{\mathtt{erel}} e'$ (so $\underline{L'_R \vdash_{\mathtt{erel}} \mathsf{let}\ \ell, x{=}e';\ s_1}$), and $C' \vdash_{\mathtt{rtyp}} e : \tau'$. The Term Weakening Lemma ensures $L; \cdot; \Gamma'_S \Gamma'_U, x{:}(\tau', \ell); \cdot; \epsilon; \tau \vdash_{\mathtt{styp}} s_1$, so $\underline{C'; \tau \vdash_{\mathtt{styp}} \mathsf{let}\ \ell, x{=}e';\ s_1}$.

If $s = \mathsf{open}\ e\ \mathsf{as}\ \ell, \alpha, x;\ s_1$, the argument is analogous to the case $s = \mathsf{let}\ \ell, x{=}e;\ s_1$ although s_1 is type-checked under a different context. Inverting the typing derivation also provides $L; \cdot \vdash_{\mathtt{k}} \ell : \mathtt{LU}$ and $L; \cdot \vdash_{\mathtt{k}} \tau : \mathtt{AU}$. So the Context Weakening Lemma ensures $L'; \cdot \vdash_{\mathtt{k}} \ell : \mathtt{LU}$ and $L'; \cdot \vdash_{\mathtt{k}} \tau : \mathtt{AU}$, which we need to derive $C'; \tau \vdash_{\mathtt{styp}} \mathsf{open}\ e'\ \mathsf{as}\ \ell, \alpha, x;\ s_1$.

If $s = \mathsf{sync}\ e\ s_1$, the argument is analogous to the case $s = \mathsf{let}\ \ell, x{=}e;\ s_1$ although s_1 is type-checked under a different context.

If $s = \mathsf{spawn}\ e_1(e_2)$ and e_1 is not a value, let $s' = \mathsf{spawn}\ e'_1(e_2)$. Inverting $\vdash_{\mathtt{j}} H_j; \mathsf{spawn}\ e_1(e_2)$ ensures $\vdash_{\mathtt{j}} H_j$ and $\vdash_{\mathtt{jf}} e_2$. Inverting $L_R \vdash_{\mathtt{srel}} \mathsf{spawn}\ e_1(e_2)$ ensures $L_R \vdash_{\mathtt{erel}} e_1$ and $\cdot \vdash_{\mathtt{erel}} e_2$. Inverting $C; \tau \vdash_{\mathtt{styp}} \mathsf{spawn}\ e_1(e_2)$ ensures $C \vdash_{\mathtt{rtyp}} e_1 : \tau_1 \xrightarrow{\emptyset} \tau_2$, $C \vdash_{\mathtt{rtyp}} e_2 : \tau_1$, and $L; \cdot \vdash_{\mathtt{k}} \tau_1 : \mathtt{AS}$. So the induction hypothesis provides $\vdash_{\mathtt{j}} H'_j; e'_1$ (so $\underline{\vdash_{\mathtt{j}} H_j; \mathsf{spawn}\ e'_1(e_2)}$), $L'_R \vdash_{\mathtt{erel}} e'_1$ (so $\underline{L'_R \vdash_{\mathtt{srel}} \mathsf{spawn}\ e'_1(e_2)}$), and $C' \vdash_{\mathtt{rtyp}} e'_1 : \tau_1 \xrightarrow{\emptyset} \tau_2$. The Term Weakening Lemma ensures $C' \vdash_{\mathtt{rtyp}} e_2 : \tau_1$ and the Context Weakening Lemma ensures $L'; \cdot \vdash_{\mathtt{k}} \tau_1 : \mathtt{AS}$, so we can derive $\underline{C'; \tau \vdash_{\mathtt{styp}} \mathsf{spawn}\ e'_1(e_2)}$.

If $s = \mathsf{spawn}\ v(e_1)$, let $s' = \mathsf{spawn}\ v(e'_1)$. Inverting $\vdash_{\mathtt{j}} H_j; \mathsf{spawn}\ v(e_1)$ ensures $\vdash_{\mathtt{jf}} v$ and $\vdash_{\mathtt{j}} H_j; e_1$ (because the Values Effectless Lemma ensures $x, v' \not\vdash_{\mathtt{e}} v$). Inverting $L_R \vdash_{\mathtt{srel}} \mathsf{spawn}\ v(e_1)$ ensures $\cdot \vdash_{\mathtt{erel}} v$ and $L_R \vdash_{\mathtt{erel}} e_1$ (because if $L_R \vdash_{\mathtt{erel}} v$, then $\cdot \vdash_{\mathtt{erel}} e_1$ and the Values Effectless Lemma ensures $L_R = \cdot$). Inverting $C; \mathsf{spawn}\ v(e_1) \vdash_{\mathtt{styp}} \tau$ ensures $C \vdash_{\mathtt{rtyp}} v : \tau_1 \xrightarrow{\emptyset} \tau_2$, $C \vdash_{\mathtt{rtyp}} e_1 : \tau_1$, and $L; \cdot \vdash_{\mathtt{k}} \tau_1 : \mathtt{AS}$. So the induction hypothesis provides $\vdash_{\mathtt{j}} H'_j; e'_1$ (so $\underline{\vdash_{\mathtt{j}} H'_j; \mathsf{spawn}\ v(e'_1)}$), $L'_R \vdash_{\mathtt{erel}} e'_1$ (so $\underline{L'_R \vdash_{\mathtt{erel}} \mathsf{spawn}\ v(e'_1)}$), and $C' \vdash_{\mathtt{rtyp}} e'_1 : \tau_1$. The Term Weakening Lemma ensures $C' \vdash_{\mathtt{rtyp}} v : \tau_1 \xrightarrow{\emptyset} \tau_2$ and the Context Weakening Lemma ensures $L'; \cdot \vdash_{\mathtt{k}} \tau_1 : \mathtt{AS}$, so we can derive $\underline{C'; \tau \vdash_{\mathtt{styp}} \mathsf{spawn}\ v(e'_1)}$.

- DS5.14: There are two cases. If $s = s_1; s_2$, let $s' = s'_1; s_2$. The case is inductive. Inverting $\vdash_{\mathtt{j}} H_j; (s_1; s_2)$ ensures $\vdash_{\mathtt{j}} H_j; s_1$ and $\vdash_{\mathtt{jf}} s_2$. Inverting $L_R \vdash_{\mathtt{srel}} s_1; s_2$ ensures $L_R \vdash_{\mathtt{srel}} s_1$ and $\cdot \vdash_{\mathtt{srel}} s_2$. Inverting $C; \tau \vdash_{\mathtt{styp}} s_1; s_2$ ensures $C; \tau \vdash_{\mathtt{styp}} s_1$ and $C; \tau \vdash_{\mathtt{styp}} s_2$. So the induction hypothesis provides $\vdash_{\mathtt{j}} H'_j; s'_1$ (so with $\vdash_{\mathtt{jf}} s_2$ we have $\underline{\vdash_{\mathtt{j}} H'_j; (s'_1; s_2)}$), $L'_R \vdash_{\mathtt{srel}} s'_1$ (so with $\cdot \vdash_{\mathtt{srel}} s_2$ we have $\underline{L'_R \vdash_{\mathtt{srel}} s'_1; s_2}$),

and $C'; \tau \vdash_{\text{styp}} s_1'$. The Term Weakening Lemma ensures $C'; \tau \vdash_{\text{styp}} s_2$, so we have $\underline{C'; \tau \vdash_{\text{styp}} s_1'; s_2}$.

If $s = s_1; \mathsf{release}\ i$, let $s' = s_1'; \mathsf{release}\ i$. Inverting $L_R \vdash_{\text{srel}} s_1; \mathsf{release}\ i$ ensures L_R has the form i, L_{R1} and $\underline{L_{R1} \vdash_{\text{srel}} s_1}$. Letting $L_{E1} = L_E, i$ and $\epsilon_1 = \epsilon \cup i$, the $\vdash_{\text{hind}}$ assumption's hypotheses let us easily derive
$\underline{\vdash_{\text{hind}} H_{XS}; H_{0S}; H_S; H_U; L; L_0; L_X; L_{R1}; L_{E1} : \Gamma_S; \Gamma_U; \epsilon_1}$.
Inverting $\vdash_{\text{j}} H_j; s_1; \mathsf{release}\ i$ ensures $\underline{\vdash_{\text{j}} H_j; s_1}$. Inverting $C; \tau \vdash_{\text{styp}} s_1; \mathsf{release}\ i$ ensures $\underline{L; \cdot; \Gamma_S \Gamma_U; \cdot; \epsilon_1; \tau \vdash_{\text{styp}} s_1}$. Applying the induction hypothesis to the underlined facts provides some H'_{XS}, H'_{0S}, H'_S, H'_U, L', L'_0, L'_{R1}, Γ'_S, and Γ'_U such that the seven conclusions hold (with L_{E1} in place of L_E and s_1' in place of s_1). Conclusions 1, 4, 5, 6, and 7 from the induction satisfy our corresponding obligations directly. Letting $L'_R = L'_{R1}, i$, Conclusion 2 from the induction ($L'_h = L'_{R1} L_{E1}$) is equivalent to $L'_h = L'_R L_E$, which is Conclusion 2 of our obligations. Conclusion 3 from the induction is $\vdash_{\text{sind}} H'_{XS}; H'_{0S}; H'_S; H'_U; L'; L'_0; L_X; L'_{R1}; L_{E1}; \tau; s_1' : \Gamma'_S; \Gamma'_U$, from which inversion ensures $\vdash_{\text{hind}} H'_{XS}; H'_{0S}; H'_S; H'_U; L'; L'_0; L_X; L'_{R1}; L_{E1} : \Gamma'_S; \Gamma'_U; \epsilon_1$, (the assumptions of which ensure $\underline{\vdash_{\text{hind}} H'_{XS}; H'_{0S}; H'_S; H'_U; L'; L'_0; L_X; L'_R; L_E : \Gamma'_S; \Gamma'_U; \epsilon}$), $L'; \cdot; \Gamma'_S \Gamma'_U; \cdot; \epsilon_1; \tau \vdash_{\text{styp}} s_1'$ (so $\underline{C'; \tau \vdash_{\text{styp}} s_1'; \mathsf{release}\ i}$), $L'_{R1} \vdash_{\text{srel}} s_1'$ (so $\underline{L'_R \vdash_{\text{srel}} s_1'; \mathsf{release}\ i}$), and $\vdash_{\text{j}} H'_j; s_1'$ (so $\underline{\vdash_{\text{j}} H'_j; s_1'; \mathsf{release}\ i}$). Conclusion 3 follows form the underlined facts.

Lemma C.17 (Type and Release Progress).

1. *If* $\vdash_{\text{sind}} H_{XS}; H_{0S}; H_S; H_U; L; L_0; L_X; L_R; L_E; \tau; s : \Gamma_S; \Gamma_U$, *then* $s = v$ *for some* v, $s = \mathsf{return}\ v$ *for some* v *or there exist* i, H', $\overline{L}'$, s_{opt}, *and* s' *such that* $H_{XS} H_{XU} H_{0S} H_S H_U; (L; L_0, j; L_R L_E); s \xrightarrow{\text{s}} H'; \overline{L}'; s_{\text{opt}}; s'$.

2. *If* $\vdash_{\text{rind}} H_{XS}; H_{0S}; H_S; H_U; L; L_0; L_X; L_R; L_E; e : \tau; \Gamma_S; \Gamma_U$, *then* $e = v$ *for some* v *or there exist* i, H', $\overline{L}'$, s_{opt}, *and* e' *such that* $H_{XS} H_{XU} H_{0S} H_S H_U; (L; L_0, j; L_R L_E); e \xrightarrow{\text{r}} H'; \overline{L}'; s_{\text{opt}}; e'$.

3. *If* $\vdash_{\text{lind}} H_{XS}; H_{0S}; H_S; H_U; L; L_0; L_X; L_R; L_E; e : \tau, \ell; \Gamma_S; \Gamma_U$, *then* $e = x$ *for some* x *or there exist* i, H', $\overline{L}'$, s_{opt}, *and* e' *such that* $H_{XS} H_{XU} H_{0S} H_S H_U; (L; L_0, j; L_R L_E); e \xrightarrow{\text{l}} H'; \overline{L}'; s_{\text{opt}}; e'$.

Proof: The proofs are by simultaneous induction on the typing derivations implied by the $\vdash_{\text{sind}}$, $\vdash_{\text{rind}}$, and $\vdash_{\text{lind}}$ assumptions, proceeding by cases on the last step in the $\vdash_{\text{styp}}$, $\vdash_{\text{rtyp}}$, or $\vdash_{\text{ltyp}}$ derivation. Throughout, let $H_j = H_S H_U$ and $C = L; \cdot; \Gamma_S \Gamma_U; \cdot; \epsilon$. Unless otherwise stated, when we apply the induction hypothesis, we use the assumed $\vdash_{\text{hind}}$ assumption unchanged and use inversion to establish the typing, release, and junk facts necessary to derive the appropriate $\vdash_{\text{sind}}$, $\vdash_{\text{rind}}$, or $\vdash_{\text{lind}}$ fact.

- SL5.1: This case is trivial because $e = x$.

- SL5.2: Let $e = *e_1$. If e_1 is a value, the Canonical Forms Lemma ensures it has the form $\&x$, so DL5.1 applies. Else inversion ensures $\vdash_{\text{j}} H_j; e_1$, $C \vdash_{\text{rtyp}} e_1 : \tau'$ for some τ', and $L_R \vdash_{\text{erel}} e_1$. So the result follows from induction and DL5.2.

- SR5.1: Let $e = x$. Inverting $C \vdash_{\text{rtyp}} x : \tau$ ensures $x \in \text{Dom}(\Gamma_S\Gamma_U)$. Inverting the $\vdash_{\text{hind}}$ assumption ensures $L; \Gamma_S \vdash_{\text{htyp}} H_{XS}H_{0S}H_S : \Gamma_S$ and $L; \Gamma_S\Gamma_U \vdash_{\text{htyp}} H_U : \Gamma_U$. So the Heap-Type Well-Formedness Lemma ensures $\text{Dom}(\Gamma_S\Gamma_U) \subseteq \text{Dom}(H)$, so DR5.1 applies.

- SR5.2: This case is analogous to case SL5.2, using DR5.3 for DL5.1 and DR5.11 for DL5.2

- SR5.3–4: Let $e = e_1.i$. If e_1 is a value, the Canonical Forms Lemma ensures it has the form (v_0, v_1), so DR5.4 applies. Else inversion ensures $\vdash_{\text{j}} H_j; e_1$, $C \vdash_{\text{rtyp}} e_1 : \tau_0 \times \tau_1$, and $L_R \vdash_{\text{erel}} e_1$. So the result follows from induction and DR5.11.

- SR5.5: This case is trivial because e is a value.

- SR5.6: Let $e = \&e_1$. If e_1 is some x, then e is a value. Else inversion ensures $\vdash_{\text{j}} H_j; e_1$, $C \vdash_{\text{ltyp}} e_1 : \tau', \ell$, and $L_R \vdash_{\text{erel}} e_1$. So the result follows from induction and DR5.10.

- SR5.7: Let $e = (e_0, e_1)$. If e_0 and e_1 are values, then e is a value. Else if e_0 is not a value, inversion ensures $\vdash_{\text{j}} H_j; e_0$, $C \vdash_{\text{rtyp}} e_0 : \tau_0$, and $L_R \vdash_{\text{erel}} e_0$. So the result follows from induction and DR5.11. Else e_0 is some v, so inversion ensures $\vdash_{\text{j}} H_j; e_1$ (because the Values Effectless Lemma ensures $x, v' \not\vdash_{\text{e}} v$), $L_R \vdash_{\text{erel}} e_1$ (because the Values Effectless Lemma ensures $L_R \vdash_{\text{erel}} v$ only if $L_R = \cdot$), and $C \vdash_{\text{rtyp}} e_1 : \tau_1$. So the result follows from induction and DR5.11.

- SR5.8: Let $e = e_1{=}e_2$. If $e_1 = x$ and $e_2 = v$, inverting $C \vdash_{\text{rtyp}} e_1{=}e_2 : \tau$ ensures $x \in \text{Dom}(\Gamma_S, \Gamma_U)$. Inverting the $\vdash_{\text{hind}}$ assumption ensures $L; \Gamma_S \vdash_{\text{htyp}} H_{XS}H_{0S}H_S : \Gamma_S$ and $L; \Gamma_S\Gamma_U \vdash_{\text{htyp}} H_U : \Gamma_U$. So the Heap-Type Well-Formedness Lemma ensures $\text{Dom}(\Gamma_S\Gamma_U) = \text{Dom}(H) - \text{Dom}(H_{XU})$, so DR5.2A applies so long as $H(x) = v'$ for some v' (i.e., $H(x) = \mathsf{junk}_{v'}$). The Values Effectless Lemma and inverting $\vdash_{\text{j}} H_j; x{=}v$ ensures $\vdash_{\text{jf}} H_j$, so it suffices to show $x \in \text{Dom}(H_U H_S)$: Because $L; \cdot \vdash_{\text{wf}} \Gamma_S\Gamma_U$, we know $(\Gamma_S\Gamma_U)(x) = (\tau, \ell)$ for some ℓ such that $L; \cdot \vdash_{\text{k}} \ell : \texttt{LU}$. So the Type Canonical Forms Lemma ensures $\ell = loc$ or $\ell = \text{S}(i)$ for some $i \in L$. If $\ell = loc$, the assumption $\vdash_{\text{shr}} \Gamma_S$ ensures

$x \in \mathrm{Dom}(\Gamma_U)$, so $L; \Gamma_S \Gamma_U \vdash_{\mathrm{htyp}} H_U : \Gamma_U$ and the Heap-Type Well-Formedness Lemma ensure $x \in \mathrm{Dom}(H_U)$. If $\ell = \mathrm{S}(i)$, inverting $C \vdash_{\mathrm{rtyp}} e_1{=}e_2 : \tau$ ensures $\cdot; \epsilon \vdash_{\mathrm{acc}} \mathrm{S}(i)$, so $i \in \epsilon$. From the $\vdash_{\mathrm{hind}}$ assumptions, that means $i \in L_E$, so $i \notin L_0$ and $i \notin L_X$. So given the $\vdash_{\mathrm{hlk}}$ assumptions, $x \notin \mathrm{Dom}(H_{XS})$ and $x \notin \mathrm{Dom}(H_{0S})$. So $x \in \mathrm{Dom}(H_j)$.

If e_1 is not some x, inversion ensures that $\vdash_{\mathrm{j}} H_j; e_1$, $C \vdash_{\mathrm{ltyp}} e_1 : \tau, \ell$, and $L_R \vdash_{\mathrm{erel}} e_1$. So the result follows from induction and DR5.10.

If e_1 is some x and e_2 is not a value, inverting $\vdash_{\mathrm{j}} H_j; e$ ensures either $e_2 = \mathsf{junk}_v$ for some v or $\vdash_{\mathrm{j}} H_j; e_2$. In the former case, $\vdash_{\mathrm{j}} H_j; e$ (that is, $\vdash_{\mathrm{j}} H_j; x{=}\mathsf{junk}_v$) ensures $H_j(x) = \mathsf{junk}_v$, so DR5.2B applies. In the latter case, inversion ensures $C \vdash_{\mathrm{rtyp}} e_2 : \tau$ and $L_R \vdash_{\mathrm{erel}} e_2$, so the result follows from induction and DR5.11.

- SR5.9: Let $e = e_1(e_2)$. If e_1 and e_2 are values, the Canonical Forms Lemma ensures e_1 has the form $(\tau_1, \ell\ x) \xrightarrow{\epsilon'} \tau_2\ s$, so DR5.5 applies. Else if e_1 is not a value, inversion ensures that $\vdash_{\mathrm{j}} H_j; e_1$, $C \vdash_{\mathrm{rtyp}} e_1 : \tau'$ for some τ', and $L_R \vdash_{\mathrm{erel}} e_1$. So the result follows from induction and DR5.11. Else e_1 is some v, so inversion ensures that $\vdash_{\mathrm{j}} H_j; e_2$ (because the Values Effectless Lemma ensures $x, v' \not\vdash_{\mathrm{e}} v$), $C \vdash_{\mathrm{rtyp}} e_2 : \tau'$ for some τ', and $L_R \vdash_{\mathrm{erel}} e_2$ (because the Values Effectless Lemma ensures $L_R \vdash_{\mathrm{erel}} v$ only if $L_R = \cdot$). So the result follows from induction and DR5.11.

- SR5.10: Let $e = \mathsf{call}\ s$. If $s = \mathsf{return}\ v$, then DR5.6 applies. Else we know $s \neq v$ because $\vdash_{\mathrm{ret}} s$. Inversion ensures that $\vdash_{\mathrm{j}} H_j; s$, $C; \tau \vdash_{\mathrm{styp}} s$, and $L_R \vdash_{\mathrm{srel}} s$. So the result follows from induction and DR5.9.

- SR5.11: Let $e = e_1[\tau]$. If e_1 is a value, the Canonical Forms Lemma ensures $e_1 = \Lambda\alpha{:}\kappa[\gamma].f$ for some f, so DR5.7 applies. Else inversion ensures $\vdash_{\mathrm{j}} H_j; e_1$, $C \vdash_{\mathrm{rtyp}} e_1 : \tau'$ for some τ', and $L_R \vdash_{\mathrm{erel}} e_1$. So the result follows from induction and DR5.11.

- SR5.12: $e = \mathsf{pack}\ \tau_1, e_1\ \mathsf{as}\ \tau_2$ If e_1 is a value, then e is a value. Else inversion ensures $\vdash_{\mathrm{j}} H_j; e_1$, $C \vdash_{\mathrm{rtyp}} e_1 : \tau'$ for some τ', and $L_R \vdash_{\mathrm{erel}} e_1$. So the result follows from induction and DR5.11.

- SR5.13–15: These cases are trivial because e is a value.

- SR5.16: This case holds vacuously because $\not\vdash_{\mathrm{j}} H_j; \mathsf{junk}_v$.

- SR5.17: This case is trivial because e is a value.

- SR5.18: Rule DR5.8 applies.

- SS5.1: Let $s = e$. If e is a value, the result is immediate. Else inversion ensures $\models_{\text{j}} H_j; e,\ C \vdash_{\text{rtyp}} e : \tau'$, and $L_R \vdash_{\text{erel}} e$. So the result follows from induction and DS5.13.

- SS5.2: This case is analogous to the previous case.

- SS5.3: Let $s = s_1; s_2$. If $s_1 = v$, DS5.2 applies. If $s_1 = \mathsf{return}\ v$, DS5.3 applies. Else inversion ensures $\models_{\text{j}} H_j; s_1,\ C; \tau \vdash_{\text{styp}} s_1$, and $L_R \vdash_{\text{srel}} s_1$. So the result follows from induction and DS5.14.

- SS5.4: Rule DS5.6 applies.

- SS5.5: Let $s = \mathsf{if}\ e\ s_1\ s_2$. If e is a value, inverting $C; \tau \vdash_{\text{styp}} \mathsf{if}\ e\ s_1\ s_2$ ensures it has type int, so the Canonical Forms Lemma ensures it is some i. So either DS5.4 or DS5.5 applies. Else inversion ensures $\models_{\text{j}} H_j; e,\ C \vdash_{\text{rtyp}} e : \text{int}$, and $L_R \vdash_{\text{erel}} e$. So the result follows from induction and DS5.13.

- SS5.6: Let $s = \mathsf{let}\ \ell, x{=}e;\ s_1$. If e is a value, DS5.1 applies. Else inversion ensures $\models_{\text{j}} H_j; e,\ C \vdash_{\text{rtyp}} e : \tau'$, and $L_R \vdash_{\text{erel}} e$. So the result follows from induction and DS5.13.

- SS5.7: Let $s = \mathsf{open}\ e\ \mathsf{as}\ \ell, \alpha, x;\ s_1$. If e is a value, inverting $C; \tau \vdash_{\text{styp}} \mathsf{open}\ e\ \mathsf{as}\ \ell, \alpha, x;\ s_1$ ensures it has an existential type, so the Canonical Forms Lemma ensures it is an existential package. So DS5.7 applies. Else inversion ensures $\models_{\text{j}} H_j; e,\ C \vdash_{\text{rtyp}} e : \tau'$, and $L_R \vdash_{\text{erel}} e$. So the result follows from induction and DS5.13.

- SS5.8: Let $s = \mathsf{sync}\ e\ s_1$. If e is a value, inverting $C; \tau \vdash_{\text{styp}} \mathsf{sync}\ e\ s_1$ ensures $C \vdash_{\text{rtyp}} e : \text{lock}(\ell)$ for some ℓ. The Typing Well-Formedness lemma ensures $L; \cdot \models_{\text{k}} \text{lock}(\ell) : \mathtt{LU}$, so the Type Canonical Forms Lemma ensures $\ell = \text{S}(i)$ or $\ell = \mathit{loc}$. In the former case, the Canonical Forms Lemma ensures $e = \mathsf{lock}\ i$ so DS5.8 applies so long as i is available. The statement of the lemma is weak enough that assuming i is available suffices. In the latter case, the Canonical Forms Lemma ensures $e = \mathsf{nonlock}$, so DS5.9 applies. Else e is not a value. Inversion ensures $\models_{\text{j}} H_j; e,\ C \vdash_{\text{rtyp}} e : \text{lock}(\ell)$, and $L_R \vdash_{\text{erel}} e$. So the result follows from induction and DS5.13.

- SS5.9: Let $s = s_1; \mathsf{release}\ i$. If $s_1 = v$ or $s_1 = \mathsf{return}\ v$ for some v, then DS5.10 or DS5.11 applies so long as $i \in L_R L_E$. Inverting $L_R \vdash_{\text{srel}} s_1; \mathsf{release}\ i$ ensures $i \in L_R$. Else inversion ensures $\models_{\text{j}} H_j; s_1,\ L; \cdot; \Gamma_S \Gamma_U; \cdot; \epsilon \cup i; \tau \vdash_{\text{styp}} s_1$, and $L_{R1} \vdash_{\text{srel}} s_1$ where $L_R = L_{R1}, i$. The result follows from induction and DS5.13 so long as $\vdash_{\text{hind}} H_{XS}; H_{0S}; H_S; H_U; L; L_0; L_X; L_{R1}; L_{E1}; \Gamma_S; \Gamma_U; \epsilon_1$ where $L_{E1} =$

L_E, i and $\epsilon_1 = \epsilon \cup i$. The $\vdash_{\text{hind}}$ assumption provides all the facts we need (note that $L_{R1}L_{E1} = L_R L_E$).

- SS5.10: Let $s = \mathsf{spawn}\ e_1(e_2)$. If e_1 and e_2 are values, DR5.12 applies. Else if e_1 is not a value, inversion ensures $\vdash_{\text{j}} H_j; e_1$, $C \vdash_{\text{rtyp}} e_1 : \tau'$ for some τ', and $L_R \vdash_{\text{erel}} e_1$. So the result follows from induction and DS5.13. Else e_1 is some v, so inversion ensures $\vdash_{\text{j}} H_j; e_2$ (because the Values Effectless Lemma ensures $x, v' \not\vdash_{\text{je}} v$), $C \vdash_{\text{rtyp}} e_2 : \tau'$ for some τ', and $L_R \vdash_{\text{erel}} e_2$ (because the Values Effectless Lemma ensures $L_R \vdash_{\text{erel}} v$ only if $L_R = \cdot$). So the result follows from induction and DS5.13.

Lemma C.18 (Preservation). *If $\vdash_{\text{prog}} P$ and $P \to P'$, then either P' has no threads or $\vdash_{\text{prog}} P'$.*

Proof: The proof is by cases on the rule used for $P \to P'$.

For case DP5.1, let $P = L; L_0; H; (L_1, s_1) \cdots (L_n, s_n)$ where i is the thread that takes a step. Inverting $\vdash_{\text{prog}} P$, the conditions for the Type and Release Preservation Lemma are satisfied by letting $H_{XS} = H_{1S} \ldots H_{(i-1)S} H_{(i+1)S} \ldots H_{nS}$, $H_{XU} = H_{1U} \ldots H_{(i-1)U} H_{(i+1)U} \ldots H_{nU}$, $H_S = H_{iS}$, $H_U = H_{iU}, L_X = L_1 \ldots L_{i-1} L_{i+1} \ldots L_n$, $L_R = L_i$, $L_E = \cdot$, $s = s_i$, and $\tau = \tau_i$. The lemma ensures
$P' = L'_0 L_X L'_R L_E; L'_0; H'_{XS} H_{XU} H'_{0S} H'_{iS} H'_U;$
$(L_1, s_1) \ldots (L_{i-1}, s_{i-1})(L'_R L_E, s')(L_{i+1}, s_{i+1}) \ldots (L_n, s_n)$ (where we write H'_{iS} where the statement of the lemma writes H'_S) and the lemma's conclusions hold. We must establish $\vdash_{\text{prog}} P'$ from these conclusions and $\vdash_{\text{prog}} P$.

We have shown $\underline{L' = L'_0 L_X L'_R L_E}$. Letting $H'_S = H'_{0S} H'_{XS} H'_{iS}$, we have shown $\underline{H' = H'_S H_{1U} \ldots H_{(i-1)U} H'_U H_{(i+1)U} \ldots H_{nU}}$. For $\underline{H'_S = H'_{0S} H'_{1S} \ldots H'_{nS}}$, it suffices to choose H'_{jS} for $j \neq i$ and $1 \leq j \leq n$ such that $H'_{XS} = H'_{1S} \ldots H'_{(i-1)S} H'_{(i+1)S} \ldots H'_{nS}$. Using Conclusion 6, choose $H'_{jS} = H_{jS}$ with one possible exception: If $H'_{XS} = H_{XS}, x \mapsto v$, then Conclusion 3 ensures $\Gamma'_S; L_X \vdash_{\text{hlk}} H'_{XS}$, so $\Gamma'_S(x) = (\tau', \mathrm{S}(k))$ for some τ' and $k \in L_X$. So $k \in L_j$ for some j. In this case, let $H'_{jS} = H_{jS}, x \mapsto v$. The $\vdash_{\text{hind}}$ assumption from Conclusion 3 provides $\underline{L'; \Gamma'_S \vdash_{\text{htyp}} H'_S : \Gamma'_S}$, $\underline{L \vdash_{\text{shr}} \Gamma_S}$, $\underline{\Gamma'_S; L'_0 \vdash_{\text{hlk}} H'_{0S}}$, and $\underline{\vdash_{\text{jf}} H_{0S}}$. The remaining obligations involve threads that are either i or some $j \neq i$. For thread i, Conclusion 3 provides all the obligations (using Γ'_U for Γ_{iU}, L'_R for L_i, H'_U for H_{iU}, etc.) except for $\vdash_{\text{ret}} s'$, which follows from inverting $\vdash_{\text{prog}} P$ and the Return Preservation Lemma. For thread $j \neq i$, the appropriate weakening lemmas and $\vdash_{\text{prog}} P$ ensure $\underline{L'; \Gamma'_S \Gamma_{jU} \vdash_{\text{htyp}} H_{jU} : \Gamma_{jU}}$, $\underline{L' \vdash_{\text{loc}} \Gamma_{jU}}$, and $\underline{L'; \cdot; \Gamma'_S \Gamma_{jU}; \cdot; \emptyset; \tau_j \vdash_{\text{styp}} s_j}$. Without need for weakening, $\vdash_{\text{prog}} P$ ensures $\underline{\vdash_{\text{ret}} s_j}$ and $\underline{L_j \vdash_{\text{srel}} s_j}$. The remaining obligations involve H'_{jS}, which could be H_{jS} or $H_{jS}, x \mapsto v$ for some x and v. In either case, $\vdash_{\text{P}}$ provides $\vdash_{\text{j}} H_{jS} H_{jU}; s_j$, so we can derive $\underline{\vdash_{\text{j}} H'_{jS} H_{jU}; s_j}$. Similarly, $\vdash_{\text{P}}$ provides $\Gamma_S; L_j \vdash_{\text{hlk}} H_{jS}$. With the Heap Weaken-

ing Lemma, this fact suffices to derive $\Gamma_S; L_j \vdash_{\text{hlk}} H'_{jS}$ so long as $\Gamma_S(x) = (\tau', \mathrm{S}(k))$ and $k \in L_j$, which is exactly why we put x in H'_{jS}.

For case DP5.2, we use the entire argument for the previous case. It then remains to establish the assumptions for the new thread, call it $n+1$. Letting $H'_{(n+1)U} = \cdot$, $H'_{(n+1)S} = \cdot$, $L'_{n+1} = \cdot$, and $\Gamma'_{(n+1)U} = \cdot$, Conclusion 7 of the Type and Release Preservation Lemma provides four of the obligations for thread $n+1$. The other three are trivial.

For case DP5.3, $(L_i, s_i) = (\cdot, \mathsf{return}\ v)$ for some i and v. If this thread is the only one, then P' has no threads and we are done. Else, the assumptions from $\vdash_{\text{prog}} P$ almost suffice to show $\vdash_{\text{prog}} P'$. Because $L_i = \cdot$, inverting $\Gamma_S; L_i \vdash_{\text{hlk}} H_{iS}$ ensures $H_{iS} = \cdot$. The complication is how to account for H_{iU} (which is in fact garbage). We take some $j \neq i$, let $H'_{jU} = H_{jU}H_{iU}$, and show $\vdash_{\text{prog}} P'$ using H'_{jU} for H_{jU}. The assumptions that are not provided immediately via $\vdash_{\text{prog}} P$ are:

1. $L; \Gamma_S\Gamma_{jU}\Gamma_{iU} \vdash_{\text{htyp}} H_{jU}H_{iU} : \Gamma_{jU}\Gamma_{iU}$
2. $L \vdash_{\text{loc}} \Gamma_{jU}, \Gamma_{iU}$
3. $L; \cdot \vdash_{\text{wf}} \Gamma_{jU}\Gamma_{iU}$
4. $L; \cdot; \Gamma_S\Gamma_{jU}\Gamma_{iU}; \cdot; \emptyset; \tau_j \vdash_{\text{styp}} s_j$
5. $\vdash_{\text{j}} H_{jS}H_{jU}H_{iU}; s_j$

The first assumption is proven by induction on the size of H_{iU}, using the assumptions that H_{iU} and H_{jU} type-check separately and the Heap Weakening Lemma. The second assumption is proven by induction on the size of Γ_{iU} using the assumptions $L \vdash_{\text{loc}} \Gamma_{jU}$ and $L \vdash_{\text{loc}} \Gamma_{iU}$. The third assumption is proven by induction on the size of Γ_{iU} using the assumptions $L; \cdot \vdash_{\text{wf}} \Gamma_{jU}$ and $L; \cdot \vdash_{\text{wf}} \Gamma_{iU}$. The fourth assumption follows from the Term Weakening Lemma. For the fifth assumption, the form of s_i, the assumption $\vdash_{\text{j}} H_{iS}H_{iU}; s_i$, and the Values Effectless Lemma ensure $\vdash_{\text{jf}} H_{iU}$. Hence the assumption $\vdash_{\text{j}} H_{jS}H_{jU}; s_j$ ensures $\vdash_{\text{j}} H_{jS}H_{jU}H_{iU}; s_j$.

Lemma C.19 (Progress). *If $\vdash_{\text{prog}} P = L; L_0; H; (L_1, s_1) \cdots (L_n, s_n)$, then for all $1 \leq i \leq n$, either $s_i = \mathsf{return}\ v$ and $L_i = \cdot$ or there exists a j such that $H; (L; L_0, j; L_i); s_i \xrightarrow{\text{s}} H'; \overline{L}'; s_{\text{opt}}; s'_i$ for some H', $\overline{L}'$, s_{opt}, and s'_i. (Note that the latter case subsumes the situation where no j needs to be added to L_0.)*

Proof: Let (L_i, s_i) be an arbitrary thread in P. By assumption, we have all the hypotheses for $\vdash_{\text{prog}} P$. The conditions for the Type and Release Progress Lemma are satisfied by letting $H_{XS} = H_{1S} \ldots H_{(i-1)S}H_{(i+1)S} \ldots H_{nS}$, $H_{XU} = H_{1U} \ldots H_{(i-1)U}H_{(i+1)U} \ldots H_{nU}$, $H_S = H_{iS}$, $H_U = H_{iU}, L_X = L_1 \ldots L_{i-1}L_{i+1} \ldots L_n$, $L_R = L_i$, $L_E = \cdot$, $s = s_i$, and $\tau = \tau_i$.

Hence one of the three cases in the conclusion of the Type and Release Progress Lemma hold. In fact, $s = v$ is impossible because Values Effectless Lemma provides $\not\vdash_{\text{ret}} v$, but we assume $\vdash_{\text{ret}} v$. If $s = \mathsf{return}\ v$, then the assumption $L_i \vdash_{\text{srel}} s$ and the Values Effectless Lemma ensure that $L_i = \cdot$. So the program can take a step with DP5.3. The remaining possibility is allowed directly by the lemma we are proving. (In this case, if no j is necessary, then either DP5.1 or DP5.2 lets the program take a step.)

Finally, we can prove the Type Soundness theorem by induction on the length of the execution sequence. First, it is trivial to establish $\vdash_{\text{prog}} \cdot; (\cdot; \cdot; \cdot); (\cdot; s)$ given the theorems assumptions, so the Progress Lemma ensures the theorem holds after 0 steps. The Preservation Lemma ensures that $\vdash_{\text{P}} P'$ if P is the state after n steps and $P \to P'$. So the Progress Lemma ensures the theorem holds after $n + 1$ steps.

Appendix D

Chapter 6 Safety Proof

This appendix proves Theorem 6.2, which we repeat here:

Definition 6.1. *State $V; H; s$ is stuck if s is not some value v, s is not* return*, and there are no V', H' and s' such that $V; H; s \xrightarrow{s} V'; H'; s'$.*

Theorem 6.2 (Type Safety). *If $V; \cdot \vdash_{\text{styp}} s : \Gamma$, $\vdash_{\text{V}} s : V$, and $V; \cdot; s \xrightarrow{s}{}^{*} V'; H'; s'$ (where $\xrightarrow{s}{}^{*}$ is the reflexive transitive closure of $\xrightarrow{s}$), then $V'; H'; s'$ is not stuck.*

Proof: The proof is by induction on the number of steps to reach $V'; H'; s'$. For zero steps, we can use the assumptions to show $\vdash_{\text{prog}} V; \cdot; s : \Gamma$. For more steps, induction and Preservation Lemma 3 (proved in this appendix) ensure $\vdash_{\text{prog}} V'; H'; s' : \Gamma$. So Progress Lemma 3 (also proved in this appendix) ensures $V'; H'; s'$ is not stuck.

Before presenting and proving the necessary lemmas in "bottom-up" order, we identify several novel aspects of the proof and then give a "top-down" overview of the proof's structure. Novel proof obligations include the following:

- Assignments to escaped locations must preserve heap typing. This result follows because for any type τ there is only one abstract rvalue r such that $\Gamma \vdash_{\text{wf}} \tau, \text{ESC}, r$.
- Preservation when $v; s$ becomes s (similarly, when we reduce if v s_1 s_2) is difficult to establish because the assumed typing of $v; s$ may use subsumption to type-check s under a weaker context than v. It is *not* the case that s necessarily type-checks under the stronger context (e.g., s may be a loop). Somewhat surprisingly, it *is* the case that the heap type-checks under an extension of the weaker context and s type-checks under this extension.
- Given $\Gamma \vdash_{\text{rtyp}} e : \tau, r, \Gamma'$, $\Gamma \vdash_{\text{htyp}} H : \Gamma'$, and $H; e \xrightarrow{r} H'; e'$ (and analogously for left-evaluation), the conventional conclusion of preservation ($\Gamma'' \vdash_{\text{rtyp}} e' :$

τ, r, Γ' for an appropriate Γ'') is not strong enough for an inductive proof. Specifically, expressions with under-specified evaluation order have $\Gamma' = \Gamma$ and preservation requires that $\Gamma'' = \Gamma$. In fact, because of subsumption, we must show that Γ'' can be Γ whenever $\Gamma \vdash \Gamma' \leq \Gamma$. Interestingly, this extended preservation result is what fails to hold if we add sequence expressions as described in Section 6.2. Under a, "permutation semantics," the result does not hold, but it does not need to for safety. Under a, "C ordering semantics," this result is necessary.

- The Weakening Lemmas for typing judgments must allow extensions to the assumed typing context to appear in the produced context. Without this extension, the result is too weak due to under-specified evaluation order. The contexts with which we can extend the assumed context are subject to technical conditions that avoid variable clashes.

- Preservation when copying a loop body s requires systematic renaming of s. We argue that the renamed copy still type-checks under the same Γ_1 and produces the same Γ because Γ_1 and Γ cannot mention variables that s allocates.

The Progress Lemma ensures well-typed program states are not stuck. As usual, some cases use the Canonical Forms Lemma to argue about the form of values. For example, a value with abstract rvalue ALL@ cannot be 0. Case ST6.4 is interesting because the derivation uses $\vdash_{\text{ltyp}}$, but we need to take a right-expression step. Subtyping Preservation Lemma 2 ensures the expression is a well-typed right-expression. Case SS6.4 must argue that it is always possible to use systematic renaming such that rule DS6.6 applies.

The Preservation Lemma ensures evaluation preserves typing. The lemmas for expressions and tests are simpler because such terms cannot extend the heap. We discussed above why expression preservation has unconventional obligations. When $*\&x$ becomes x, we need the Subtyping Preservation Lemma to show that any subsumption used in the typing derivation for $*\&x$ can be duplicated when typing x. Case SR6.10 (assignment) is particularly complicated when e has the form $x{=}v$. When the assignment changes the abstract rvalue for x, we use the Assignment Preservation Lemma to argue that the rest of the heap (i.e., locations other than x) continue to type-check. For type-checking the contents of x, we use Canonical Forms Lemma 8. We also use Heap Subsumption Lemma 3 to show that if the new abstract rvalue of x is less approximate than the old one, then we can subsume the heap to its old type. (Intuitively, we need to do so when assignments are nested within under-specified evaluation-order expressions.) When the assignment is to an escaped location, we argue that the heap does not

change type. Cases SR6.12 and SL6.3 need Abstract-Ordering Transitivity Lemma 5, which states that the ordering relationship on typing contexts is transitive.

The only interesting case for test-expression preservation is ST6.4 (which uses a run-time test to refine typing information) when the expression is some x. Intuitively, we argue by cases on the form of $H(x)$ that ST6.1 or ST6.3 let us derive a typing with the refined type information, using $H(x)$ in place of x. The Assignment Preservation Lemma ensures the rest of the heap still type-checks under the refined information. When $H(x) = \&y$ for some y, we use Values Effectless Lemma 2 to ensure $\Gamma_0 \vdash \&y \leq \text{ALL@}$. This fact and some simple observations about well-formed typing contexts (the Typing Well-Formedness Lemma) and ordering judgments (the Abstract-Ordering Inversion Lemma) let us derive $\Gamma_1 \vdash_{\text{rtyp}} \&y : \tau, \text{ALL@}, \Gamma_1$ if y is escaped. We use some technical lemmas to show this fact, but intuitively it follows because x originally had abstract rvalue ALL$*$.

Preservation for statements must account for evaluation steps that allocate memory or make (renamed) copies of loop bodies. Case SS6.1 uses Preservation Lemma 1. Case SS6.2 is trivial.

Case SS6.3 is surprisingly complicated. If the statement has the form $v; s$, then the Value Elimination Lemma provides the interesting results. In turn, this lemma uses the Heap Subsumption Lemma to handle any subsumption that the typing of v introduced. The Values Effectless Lemma ensures the typing of v changes the typing context only via subsumption. If the statement has the form $\mathsf{return}; s$, then an inordinate amount of bookkeeping is necessary to prove we can produce the same typing context with return as with $\mathsf{return}; s$. For part of the argument, we need Weakening Lemma 3 to ensure a well-formed context that we can then restrict with SS6.8. Finally the statement may have the form $s_1; s_2$ and s_1 becomes s_1'. We need Weakening Lemma 11 to argue that s_2 still type-checks. Interestingly, we do *not* need Weakening Lemma 9. Intuitively, the typing of s_1' can use subsumption to produce the same typing context as s_1.

Case SS6.4 is the only case that must argue about systematic renaming. Copying the loop body increases the number of variables allocated in the statement, but the assumptions for $\vdash_{\text{prog}}$ and rule DS6.6 sufficiently restrict what new variables are used. The Systematic Renaming Lemma ensures the renamed body type-checks with renamed typing contexts. The restrictions on the renaming ensure the typing contexts do not mention variables that the body binds, so the Useless Renaming Lemma ensures the renamed body type-checks under unchanged typing contexts. We also need Weakening Lemma 10 to show that the test type-checks even though new variables have been introduced.

Case SS6.5 uses either the Value Elimination Lemma like case SS6.3 or Preservation Lemma 2 like case SS6.1 uses Preservation Lemma 1. Case SS6.6 allocates memory. Weakening Lemma 7 ensures the heap still type-checks. Cases SS6.7 and

SS6.8 follow from induction.

We now note some interesting arguments from the proofs of the auxiliary lemmas.

The proof of Assignment Preservation Lemmas 4 and 5 must establish that induction applies when the assumed typing derivation ends with SL6.3 or SR6.12. The Values Effectless Lemma ensures the shorter derivation produces a weaker context and the assumptions of SL6.3 or SR6.12 ensure it produces a stronger context. Hence the Abstract-Ordering Antisymmetry Lemma ensures it produces the same context it consumes, so the induction hypothesis applies.

The Abstract-Ordering Antisymmetry Lemma is also crucial for cases of the Subtyping Preservation Lemma and Canonical Forms Lemma that have derivations ending with SR6.13, which subsumes abstract rvalues.

The Value Elimination Lemma proof uses the Values Effectless Lemma and Heap Subsumption Lemma to show that the assumed heap type-checks under a weaker context suitably extended. To show that the assumed statement type-checks under the extension, we need Weakening Lemma 9, which is complicated only because of renaming issues. (Compare it with Weakening Lemma 7.) The proof of Weakening Lemma 9 requires Weakening Lemmas 1–8.

The Heap Subsumption Lemma proof uses the Abstract-Ordering Antisymmetry Lemma to dismiss complications due to SL6.3 and SR6.12.

The Values Effectless Lemma proof needs the Abstract-Ordering Transitivity Lemma to ensure multiple subsumption steps produce only successively weaker results.

The Abstract-Ordering Inversion Lemma and Typing Well-Formedness Lemma make rather obvious technical points needed throughout other proofs.

Lemma D.1 (Typing Well-Formedness).

1. *If* $\Gamma_0 \vdash_{\text{wf}} \Gamma_1$ *and* $\text{Dom}(\Gamma_0) = \text{Dom}(\Gamma_2)$, *then* $\Gamma_2 \vdash_{\text{wf}} \Gamma_1$.

2. *If* $\Gamma_0 \vdash \Gamma_1 \leq \Gamma_2$, *then* $\text{Dom}(\Gamma_1) = \text{Dom}(\Gamma_2)$.

3. *If* $\Gamma_0 \vdash_{\text{ltyp}} e : \tau, \ell, \Gamma_1$ *and* $\Gamma_0 \vdash_{\text{wf}} \Gamma_0$, *then* $\text{Dom}(\Gamma_0) = \text{Dom}(\Gamma_1)$ *and* $\Gamma_1 \vdash_{\text{wf}} \Gamma_1$.
   If $\Gamma_0 \vdash_{\text{rtyp}} e : \tau, r, \Gamma_1$ *and* $\Gamma_0 \vdash_{\text{wf}} \Gamma_0$, *then* $\text{Dom}(\Gamma_0) = \text{Dom}(\Gamma_1)$ *and* $\Gamma_1 \vdash_{\text{wf}} \Gamma_1$.

4. *If* $V; \Gamma_0 \vdash_{\text{tst}} e : \Gamma_1; \Gamma_2$ *and* $\Gamma_0 \vdash_{\text{wf}} \Gamma_0$, *then*
   $\text{Dom}(\Gamma_0) \subseteq \text{Dom}(\Gamma_1) \subseteq \text{Dom}(\Gamma_0) \cup V$, $\Gamma_1 \vdash_{\text{wf}} \Gamma_1$,
   $\text{Dom}(\Gamma_0) \subseteq \text{Dom}(\Gamma_2) \subseteq \text{Dom}(\Gamma_0) \cup V$, *and* $\Gamma_2 \vdash_{\text{wf}} \Gamma_2$.

5. *If* $V; \Gamma_0 \vdash_{\text{styp}} s : \Gamma_1$, $\vdash_{\text{V}} s : V'$, $V' \subseteq V$, *and* $\Gamma_0 \vdash_{\text{wf}} \Gamma_0$, *then*
   $\text{Dom}(\Gamma_1) \subseteq \text{Dom}(\Gamma_0) \cup V$ *and* $\Gamma_1 \vdash_{\text{wf}} \Gamma_1$.

Proof:

1. By induction on the assumed derivation and inspection of the rules for $\Gamma \vdash_{\text{wf}} \tau, k, r$

2. By induction on the assumed derivation

3. The proof is by simultaneous induction on the assumed typing derivations. Cases SR6.1–6 are trivial because $\Gamma_1 = \Gamma_0$. Cases SR6.7A–B and SR6.8A–D follow from induction. Case SR6.9 is trivial because $\Gamma_1 = \Gamma_0$. Case SR6.10 follows from inspection of the rules for $\vdash_{\text{aval}}$ because Γ_1 differs from Γ_0 for at most one variable, and $\vdash_{\text{aval}}$ has the necessary well-formedness hypothesis. Case SR6.11 follows from induction. Case SR6.12 follows from the previous lemma. Case SR6.13 follows from induction. Case SL6.1 is trivial because $\Gamma_1 = \Gamma_0$. Cases SL6.2A–B follow from induction. Case SL6.3 follows from the previous lemma. Case SL6.4 follows from induction.

4. The proof is by cases on the assumed typing derivation. Cases ST6.1–3 follow from the previous lemma and inspection of the rules for $V_1; V_2 \vdash_{\text{wf}} \Gamma$. Case ST6.4 follows from the previous lemma and the fact that for any Γ and τ, if $\Gamma \vdash_{\text{wf}} \tau, \text{UNESC}, \text{ALL}*$, then $\Gamma \vdash_{\text{wf}} \tau, \text{UNESC}, \text{ALL}@$ and $\Gamma \vdash_{\text{wf}} \tau, \text{UNESC}, 0$. Case ST6.5 follows from the previous lemma.

5. The proof is by induction on the assumed typing derivation. Case SS6.1 follows from Typing Well-Formedness Lemma 3. Case SS6.2 follows from inspection of the rules for $V_1; V_2 \vdash_{\text{wf}} \Gamma$. Case SS6.3 follows from two invocations of the induction hypothesis, inversion of $\vdash_{\text{V}} s_1; s_2 : V'$, and the transitivity of $\subseteq$. Case SS6.4 follows from the previous lemma. Case SS6.5 follows from the previous lemma, induction (applied to either branch), inversion of $\vdash_{\text{V}} \mathsf{if}\ e\ s_1\ s_2 : V'$, and the transitivity of $\subseteq$. Case SS6.6 follows because $\Gamma_1 \vdash_{\text{wf}} \tau, \text{UNESC}, \text{NONE}$ and inverting $\vdash_{\text{V}} s : V'$ shows $x \in V'$. Case SS6.7 follows from induction and Typing Well-Formedness Lemma 2. Case SS6.8 follows from induction and the transitivity of $\subseteq$.

Lemma D.2 (Weakening). *Suppose* $\Gamma_0 \vdash_{\text{wf}} \Gamma_0$.

1. *If* $\Gamma_0 \vdash_{\text{wf}} \ell$, *then* $\Gamma_0\Gamma_1 \vdash_{\text{wf}} \ell$.

2. *If* $\Gamma_0 \vdash_{\text{wf}} \tau, k, r$, *then* $\Gamma_0\Gamma_1 \vdash_{\text{wf}} \tau, k, r$.

3. *If* $\Gamma_0 \vdash_{\text{wf}} \Gamma_1$, *then* $\Gamma_0\Gamma_2 \vdash_{\text{wf}} \Gamma_1$. *If* $\Gamma_0 \vdash_{\text{wf}} \Gamma_1$ *and* $\Gamma_0 \vdash_{\text{wf}} \Gamma_2$, *then* $\Gamma_0 \vdash_{\text{wf}} \Gamma_1\Gamma_2$.

4. *If* $\Gamma_0 \vdash \ell_1 \leq \ell_2$, *then* $\Gamma_0\Gamma_1 \vdash \ell_1 \leq \ell_2$.

5. *If* $\Gamma_0 \vdash r_1 \leq r_2$, *then* $\Gamma_0\Gamma_1 \vdash r_1 \leq r_2$.

6. *If* $\Gamma_1 \vdash \Gamma_0 \leq \Gamma_1$, *then* $\Gamma_1\Gamma_2 \vdash \Gamma_0\Gamma_2 \leq \Gamma_1\Gamma_2$.

7. *If* $\Gamma_0 \vdash_{\text{ltyp}} e : \tau, \ell, \Gamma_1$ *and* $\Gamma_0\Gamma_2 \vdash_{\text{wf}} \Gamma_2$, *then* $\Gamma_0\Gamma_2 \vdash_{\text{ltyp}} e : \tau, \ell, \Gamma_1\Gamma_2$.
   If $\Gamma_0 \vdash_{\text{rtyp}} e : \tau, r, \Gamma_1$ *and* $\Gamma_0\Gamma_2 \vdash_{\text{wf}} \Gamma_2$, *then* $\Gamma_0\Gamma_2 \vdash_{\text{rtyp}} e : \tau, r, \Gamma_1\Gamma_2$.

8. *If* $V; \Gamma_0 \vdash_{\text{tst}} e : \Gamma_{1\text{A}}; \Gamma_{1\text{B}}$, $\Gamma_0\Gamma_2 \vdash_{\text{wf}} \Gamma_2$, *and* $V \cap \text{Dom}(\Gamma_2) = \emptyset$, *then* $V; \Gamma_0\Gamma_2 \vdash_{\text{tst}} e : \Gamma_{1\text{A}}\Gamma_2; \Gamma_{1\text{B}}\Gamma_2$.

9. *Suppose* $\Gamma_0\Gamma_2 \vdash_{\text{wf}} \Gamma_2$, $V_0 \cap \text{Dom}(\Gamma_0\Gamma_2) = \emptyset$, $\text{Dom}(\Gamma_0) \subseteq \text{Dom}(\Gamma_3) \subseteq \text{Dom}(\Gamma_0) \cup V_0$, *and* $V_2 \subseteq V_1 \subseteq V_0$. *If* $\Gamma_3 \vdash_{\text{wf}} \Gamma_3$, $V_1; \Gamma_3 \vdash_{\text{styp}} s : \Gamma_1$ *and* $\vdash_{\text{V}} s : V_2$, *then* $V_1; \Gamma_3\Gamma_2 \vdash_{\text{styp}} s : \Gamma_1$. *Furthermore, if* $\text{Dom}(\Gamma_1) \supseteq \text{Dom}(\Gamma_0)$, *then* $V_1; \Gamma_3\Gamma_2 \vdash_{\text{styp}} s : \Gamma_1\Gamma_2$.

10. *If* $V; \Gamma_0 \vdash_{\text{tst}} e : \Gamma_{1\text{A}}; \Gamma_{1\text{B}}$, *then* $V \cup V'; \Gamma_0 \vdash_{\text{tst}} e : \Gamma_{1\text{A}}; \Gamma_{1\text{B}}$.

11. *If* $V; \Gamma_0 \vdash_{\text{styp}} s : \Gamma_1$, *then* $V \cup V'; \Gamma_0 \vdash_{\text{styp}} s : \Gamma_1$.

Proof:

1. By inspection of the assumed derivation

2. By inspection of the assumed derivation

3. The proof of both statements is by induction on the size of Γ_1. The first proof uses the previous lemma.

4. By inspection of the rules for $\Gamma \vdash \ell_1 \leq \ell_2$

5. By induction on the derivation of $\Gamma_0 \vdash r_1 \leq r_2$

6. The proof is by induction on the size of Γ_2. It is a trivial consequence of the previous lemma, $\vdash k \leq k$, and $\Gamma' \vdash r \leq r$.

7. The proof is by simultaneous induction on the assumed typing derivations. Cases SR6.1–SR6.6 are trivial. Cases SR6.7A–B, SR6.8A–D, and SR6.9 follow from induction. Case SR6.10 follows from induction and Weakening Lemma 2. Case SR6.11 follows from induction. Case SR6.12 follows from induction and Weakening Lemmas 3 and 6. Case SR6.13 follows from induction and Weakening Lemma 5. Case SL6.1 is trivial. Cases SL6.2A–B follow from induction. Case SL6.3 follows from induction and Weakening Lemmas 3 and 6. Case SL6.4 follows from induction and Weakening Lemma 4.

8. The proof is by cases on the assumed typing derivation:

- ST6.1: Inversion ensures $\Gamma_0 \vdash_{\text{rtyp}} e : \tau, 0, \Gamma_{1B}$, $\Gamma_{1A} \vdash_{\text{wf}} \Gamma_{1A}$, and $\text{Dom}(\Gamma_0) \subseteq \text{Dom}(\Gamma_{1A}) \subseteq \text{Dom}(\Gamma_0) \cup V$. So the previous lemma ensures $\Gamma_0\Gamma_2 \vdash_{\text{rtyp}} e : \tau, 0, \Gamma_{1B}\Gamma_2$. By the transitivity of $\subseteq$, $\text{Dom}(\Gamma_0\Gamma_2) \subseteq \text{Dom}(\Gamma_{1A}\Gamma_2) \subseteq \text{Dom}(\Gamma_0\Gamma_2) \cup V$ (where $V \cap \text{Dom}(\Gamma_2) = \emptyset$, $\text{Dom}(\Gamma_0) \cap \text{Dom}(\Gamma_2) = \emptyset$, and $\text{Dom}(\Gamma_{1A}) \subseteq \text{Dom}(\Gamma_0) \cup V$ ensure we can write $\Gamma_{1A}\Gamma_2$). Furthermore, $\text{Dom}(\Gamma_0) \subseteq \text{Dom}(\Gamma_{1A})$, $\Gamma_0\Gamma_2 \vdash_{\text{wf}} \Gamma_2$, Weakening Lemma 3 and Typing Well-Formedness Lemma 1 ensure $\Gamma_{1A}\Gamma_2 \vdash_{\text{wf}} \Gamma_2$. With $\Gamma_{1A} \vdash_{\text{wf}} \Gamma_{1A}$, Weakening Lemma 3 ensures that means $\Gamma_{1A}\Gamma_2 \vdash_{\text{wf}} \Gamma_{1A}\Gamma_2$. So we can conclude $V; \text{Dom}(\Gamma_0\Gamma_2) \vdash_{\text{wf}} \Gamma_{1A}\Gamma_2$. The underlined results let us use ST6.1 to derive $V; \Gamma_0\Gamma_2 \vdash_{\text{tst}} e : \Gamma_{1A}\Gamma_2; \Gamma_{1B}\Gamma_2$.
- ST6.2–3: These cases are identical to each other and analogous to case ST6.1, swapping the roles of Γ_{1A} and Γ_{1B}.
- ST6.4–5: These cases follow from the previous lemma. After applying the lemma, we can use ST6.4 or ST6.5, respectively, to conclude $V; \Gamma_3\Gamma_2 \vdash_{\text{tst}} e : \Gamma_{1A}\Gamma_2; \Gamma_{1B}\Gamma_2$.

9. The proof is by induction on the assumed typing derivation, but we first derive some results that hold in all cases. Because $\text{Dom}(\Gamma_3) \subseteq \text{Dom}(\Gamma_0) \cup V_0$ and $V_0 \cap \text{Dom}(\Gamma_0\Gamma_2) = \emptyset$, we know $\text{Dom}(\Gamma_3) \cap \text{Dom}(\Gamma_2) = \emptyset$, so we can write $\Gamma_3\Gamma_2$. Furthermore, $\text{Dom}(\Gamma_0) \subseteq \text{Dom}(\Gamma_3)$, $\Gamma_0\Gamma_2 \vdash_{\text{wf}} \Gamma_2$, Weakening Lemma 3 and Typing Well-Formedness Lemma 1 ensure $\Gamma_3\Gamma_2 \vdash_{\text{wf}} \Gamma_2$. Therefore, $\Gamma_3 \vdash_{\text{wf}} \Gamma_3$ ensures $\Gamma_3\Gamma_2 \vdash_{\text{wf}} \Gamma_3\Gamma_2$.

   Typing Well-Formedness Lemma 5 ensures $\text{Dom}(\Gamma_1) \subseteq \text{Dom}(\Gamma_3) \cup V_1$ and $\Gamma_1 \vdash_{\text{wf}} \Gamma_1$. Therefore, $\text{Dom}(\Gamma_3) \subseteq \text{Dom}(\Gamma_0) \cup V_0$ and $V_1 \subseteq V_0$ ensures $\text{Dom}(\Gamma_1) \subseteq \text{Dom}(\Gamma_0) \cup V_0$. Therefore, because $V_0 \cap \text{Dom}(\Gamma_0\Gamma_2) = \emptyset$, we know $\text{Dom}(\Gamma_1) \cap \text{Dom}(\Gamma_2) = \emptyset$, so we can write $\Gamma_1\Gamma_2$. Furthermore, if $\text{Dom}(\Gamma_1) \supseteq \text{Dom}(\Gamma_0)$, then Weakening Lemma 3, Typing Well-Formedness Lemma 1, and $\Gamma_0\Gamma_2 \vdash_{\text{wf}} \Gamma_2$ ensure $\Gamma_1\Gamma_2 \vdash_{\text{wf}} \Gamma_2$. Therefore, $\Gamma_1 \vdash_{\text{wf}} \Gamma_1$ ensures $\Gamma_1\Gamma_2 \vdash_{\text{wf}} \Gamma_1\Gamma_2$.

   - SS6.1: By inversion, $\Gamma_3 \vdash_{\text{rtyp}} e : \tau, r, \Gamma_1$. So Weakening Lemma 7 ensures $\Gamma_3\Gamma_2 \vdash_{\text{rtyp}} e : \tau, r, \Gamma_1\Gamma_2$, so SS6.1 lets us derive $V_1; \Gamma_3\Gamma_2 \vdash_{\text{styp}} e : \Gamma_1\Gamma_2$. Because $\Gamma_1 \vdash_{\text{wf}} \Gamma_1$, SS6.8 lets us derive $V_1; \Gamma_3\Gamma_2 \vdash_{\text{styp}} e : \Gamma_1$.
   - SS6.2: By inversion, $\text{Dom}(\Gamma_3) \subseteq \text{Dom}(\Gamma_1) \subseteq \text{Dom}(\Gamma_3) \cup V_1$. Therefore, $\text{Dom}(\Gamma_0) \subseteq \text{Dom}(\Gamma_3)$ ensures $\text{Dom}(\Gamma_0) \subseteq \text{Dom}(\Gamma_1)$. So we argued above that $\Gamma_1\Gamma_2 \vdash_{\text{wf}} \Gamma_1\Gamma_2$. By transitivity of $\subseteq$, we have $\text{Dom}(\Gamma_3\Gamma_2) \subseteq \text{Dom}(\Gamma_1\Gamma_2) \subseteq \text{Dom}(\Gamma_3\Gamma_2) \cup V_1$. So SS6.2 lets us derive $V_1; \Gamma_3\Gamma_2 \vdash_{\text{styp}} \mathsf{return} : \Gamma_1\Gamma_2$. Because $\Gamma_1 \vdash_{\text{wf}} \Gamma_1$, SS6.8 lets us derive $V_1; \Gamma_3\Gamma_2 \vdash_{\text{styp}} \mathsf{return} : \Gamma_1$.

- SS6.3: By inversion, $V_1; \Gamma_3 \vdash_{\text{styp}} s_1 : \Gamma'$, $\underline{V_1 - V'; \Gamma' \vdash_{\text{styp}} s_2 : \Gamma_1}$, and $\vdash_{\text{V}} s_1 : V'$ for some Γ' and V'. Inversion of $\vdash_{\text{V}} s_1; s_2 : V_2$ and the transitivity of $\subseteq$ ensures $\underline{\vdash_{\text{V}} s_2 : V_2 - V'}$ and $V' \subseteq V_2$. Because $V_1; \Gamma_3 \vdash_{\text{styp}} s_1 : \Gamma'$ and $V' \subseteq V_2$, induction ensures $V_1; \Gamma_3\Gamma_2 \vdash_{\text{styp}} s_1 : \Gamma'$. So SS6.3 lets us derive $V_1; \Gamma_3\Gamma_2 \vdash_{\text{styp}} s_1; s_2 : \Gamma_1$, as required.

  Now suppose $\text{Dom}(\Gamma_1) \supseteq \text{Dom}(\Gamma_0)$. Typing Well-Formedness Lemma 5 and $V_1 - V'; \Gamma' \vdash_{\text{styp}} s_2 : \Gamma_1$ ensure $\text{Dom}(\Gamma_1) \subseteq \text{Dom}(\Gamma') \cup (V_1 - V')$. Therefore, $\text{Dom}(\Gamma_0) \subseteq \text{Dom}(\Gamma') \cup (V_1 - V')$. Because $V_0 \cap \text{Dom}(\Gamma_0\Gamma_2) = \emptyset$ and $(V_1 - V') \subseteq V_0$, we know $(V_1 - V') \cap \text{Dom}(\Gamma_0) = \emptyset$. Therefore, $\underline{\text{Dom}(\Gamma_0) \subseteq \text{Dom}(\Gamma')}$. Therefore, $V_1; \Gamma_3 \vdash_{\text{styp}} s_1 : \Gamma'$, $V' \subseteq V_2$, and induction ensure $V_1; \Gamma_3\Gamma_2 \vdash_{\text{styp}} s_1 : \Gamma'\Gamma_2$.

  Typing Well-Formedness Lemma 5 and $V_1; \Gamma_3 \vdash_{\text{styp}} s_1 : \Gamma'$ ensure $\underline{\Gamma' \vdash_{\text{wf}} \Gamma'}$ and $\text{Dom}(\Gamma') \subseteq \text{Dom}(\Gamma_3) \cup V_1$. Therefore, $\text{Dom}(\Gamma_3) \subseteq \text{Dom}(\Gamma_0) \cup V_0$ and $V_1 \subseteq V_0$ ensure $\underline{\Gamma' \subseteq \text{Dom}(\Gamma_0) \cup V_0}$. Furthermore, $(V_2 - V') \subseteq V_2$ and $V_2 \subseteq V_1 \subseteq V_0$ ensures $\underline{(V_2 - V') \subseteq (V_1 - V') \subseteq V_0}$. The underlined results and induction let us conclude $V_1 - V'; \Gamma'\Gamma_2 \vdash_{\text{styp}} s_2 : \Gamma_1\Gamma_2$. We already showed $V_1; \Gamma_3\Gamma_2 \vdash_{\text{styp}} s_1 : \Gamma'\Gamma_2$ and $\vdash_{\text{V}} s_1 : V'$. So SS6.3 lets us derive $V_1; \Gamma_3\Gamma_2 \vdash_{\text{styp}} s_1; s_2 : \Gamma_1\Gamma_2$, as required.

- SS6.4: By inversion, $V_1; \Gamma_3 \vdash_{\text{tst}} e : \Gamma'; \Gamma_1$ and $V_1; \Gamma' \vdash_{\text{styp}} s_1 : \Gamma_3$ for some Γ'. Because $\Gamma_3\Gamma_2 \vdash_{\text{wf}} \Gamma_2$ and $V_1 \subseteq V_0$, the previous lemma ensures $V_1; \Gamma_3\Gamma_2 \vdash_{\text{tst}} e : \Gamma'\Gamma_2; \Gamma_1\Gamma_2$.

  Assume we could show $V_1; \Gamma'\Gamma_2 \vdash_{\text{styp}} s_1 : \Gamma_3\Gamma_2$. Then SS6.4 would let us derive $V_1; \Gamma_3\Gamma_2 \vdash_{\text{styp}} \mathsf{while}\ e\ s_1 : \Gamma_1\Gamma_2$. So SS6.8 would then let us derive $V_1; \Gamma_3\Gamma_2 \vdash_{\text{styp}} \mathsf{while}\ e\ s_1 : \Gamma_1$ because $\Gamma_1 \vdash_{\text{wf}} \Gamma_1$. So it suffices to show $V_1; \Gamma'\Gamma_2 \vdash_{\text{styp}} s_1 : \Gamma_3\Gamma_2$.

  We know $\underline{V_1; \Gamma' \vdash_{\text{styp}} s_1 : \Gamma_3}$. Inverting $\vdash_{\text{V}} \mathsf{while}\ e\ s_1 : V_2$ ensures $\underline{\vdash_{\text{V}} s_1 : V_2}$. Because $\overline{V_1; \Gamma_3 \vdash_{\text{tst}} e : \Gamma'; \Gamma_1}$, Typing Well-Formedness Lemma 4 ensures $\text{Dom}(\Gamma_3) \subseteq \text{Dom}(\Gamma') \subseteq \text{Dom}(\Gamma_3) \cup V_1$ and $\underline{\Gamma' \vdash_{\text{wf}} \Gamma'}$. Because $\text{Dom}(\Gamma_0) \subseteq \text{Dom}(\Gamma_3) \subseteq \text{Dom}(\Gamma_0) \cup V_0$ and $V_1 \subseteq V_0$, that means $\underline{\text{Dom}(\Gamma_0) \subseteq \text{Dom}(\Gamma') \subseteq \text{Dom}(\Gamma_0) \cup V_0}$. Because $\Gamma_3 \supseteq \Gamma_0$, the result we need, $V; \Gamma'\Gamma_2 \vdash_{\text{styp}} s_1 : \Gamma_3\Gamma_2$, follows from induction, using the underlined results.

- SS6.5: By inversion, $V_1; \Gamma_3 \vdash_{\text{tst}} e : \Gamma'_1; \Gamma'_2$, $\underline{V_1; \Gamma'_1 \vdash_{\text{styp}} s_1 : \Gamma_1}$, and $V_1; \Gamma'_2 \vdash_{\text{styp}} s_2 : \Gamma_1$ for some Γ'_1 and Γ'_2. Because $\Gamma_3\Gamma_2 \vdash_{\text{wf}} \Gamma_2$ and $V_1 \subseteq V_0$, the previous lemma ensures $V_1; \Gamma_3\Gamma_2 \vdash_{\text{tst}} e : \Gamma'_1\Gamma_2; \Gamma'_2\Gamma_2$.

  Because $V_1; \Gamma_3 \vdash_{\text{tst}} e : \Gamma'_1; \Gamma'_2$, Typing Well-Formedness Lemma 4 ensures $\text{Dom}(\Gamma_3) \subseteq \text{Dom}(\Gamma'_1) \subseteq \text{Dom}(\Gamma_3) \cup V_1$ and $\underline{\Gamma'_1 \vdash_{\text{wf}} \Gamma'_1}$. Therefore, $\text{Dom}(\Gamma_0) \subseteq \text{Dom}(\Gamma_3) \subseteq \text{Dom}(\Gamma_0) \cup V_1$ ensures

$\underline{\mathrm{Dom}(\Gamma_0) \subseteq \mathrm{Dom}(\Gamma_1') \subseteq \mathrm{Dom}(\Gamma_0) \cup V_1}$. Inverting $\vdash_{\mathrm{V}} s : V_2$ ensures $\underline{\vdash_{\mathrm{V}} s_1 : V'}$ and $V' \subseteq V_2$. So $V_2 \subseteq V_1 \subseteq V_0$ ensures $V' \subseteq V_1 \subseteq V_0$). So applying induction to the underlined results ensures $V_1; \Gamma_1'\Gamma_2 \vdash_{\mathrm{styp}} s_1 : \Gamma_1$ and if $\mathrm{Dom}(\Gamma_1) \supseteq \mathrm{Dom}(\Gamma_0)$, then $V_1; \Gamma_1'\Gamma_2 \vdash_{\mathrm{styp}} s_1 : \Gamma_1\Gamma_2$.

By an analogous argument, $V_1; \Gamma_2'\Gamma_2 \vdash_{\mathrm{styp}} s_2 : \Gamma_1$ and if $\mathrm{Dom}(\Gamma_1) \supseteq \mathrm{Dom}(\Gamma_0)$, then $V_1; \Gamma_2'\Gamma_2 \vdash_{\mathrm{styp}} s_2 : \Gamma_1\Gamma_2$. So we can use SS6.5 to derive the results we need.

- SS6.6: Rule SS6.6 lets derive $V_1; \Gamma_3\Gamma_2 \vdash_{\mathrm{styp}} \tau\ x : \Gamma_3\Gamma_2, x{:}\tau, \textsc{unesc}, \textsc{none}$ if $x \notin \mathrm{Dom}(\Gamma_2)$. (We already know $\Gamma_3\Gamma_2 \vdash_{\mathrm{wf}} \Gamma_3\Gamma_2$ and the assumed derivation implies $x \notin \Gamma_3$.) The assumption $\vdash_{\mathrm{V}} \tau\ x : V_2$ ensures $x \in V_2$, so the assumptions $V_0 \cap \mathrm{Dom}(\Gamma_0\Gamma_2) = \emptyset$ and $V_2 \subseteq V_1 \subseteq V_0$ suffice. Because $\Gamma_1 \vdash_{\mathrm{wf}} \Gamma_1$ (where $\Gamma_1 = \Gamma_3, x{:}\tau, \textsc{unesc}, \textsc{none}$), SS6.8 lets us derive $V_1; \Gamma_3\Gamma_2 \vdash_{\mathrm{styp}} \tau\ x : \Gamma_1$.
- SS6.7: By inversion, $V_1; \Gamma_3 \vdash_{\mathrm{styp}} s : \Gamma'$ and $\Gamma_1 \vdash \Gamma' \leq \Gamma_1$ for some Γ'. By induction, $V_1; \Gamma_3\Gamma_2 \vdash_{\mathrm{styp}} s : \Gamma'$, and if $\mathrm{Dom}(\Gamma') \supseteq \mathrm{Dom}(\Gamma_0)$, then $V_1; \Gamma_3\Gamma_2 \vdash_{\mathrm{styp}} s : \Gamma'\Gamma_2$. So SS6.7 lets us derive $V_1; \Gamma_3\Gamma_2 \vdash_{\mathrm{styp}} s : \Gamma_1$.

  Now suppose $\mathrm{Dom}(\Gamma_1) \supseteq \mathrm{Dom}(\Gamma_0)$. Then $V_1; \Gamma_3\Gamma_2 \vdash_{\mathrm{styp}} s : \Gamma'\Gamma_2$ because $\Gamma_1 \vdash \Gamma' \leq \Gamma_1$ and Typing Well-Formedness Lemma 2 ensure $\mathrm{Dom}(\Gamma_1) = \mathrm{Dom}(\Gamma')$. Typing Well-Formedness Lemma 5 ensures $\Gamma'\Gamma_2 \vdash_{\mathrm{wf}} \Gamma'\Gamma_2$. Weakening Lemma 6 and $\Gamma_1 \vdash \Gamma' \leq \Gamma_1$ ensure $\Gamma_1\Gamma_2 \vdash \Gamma'\Gamma_2 \leq \Gamma_1\Gamma_2$. So SS6.7 lets us derive $V_1; \Gamma_3\Gamma_2 \vdash_{\mathrm{styp}} s : \Gamma_1\Gamma_2$.
- SS6.8: By inversion, $V_1; \Gamma_3 \vdash_{\mathrm{styp}} s : \Gamma'\Gamma_1$ for some Γ'. By induction, $V_1; \Gamma_3\Gamma_2 \vdash_{\mathrm{styp}} s : \Gamma'\Gamma_1$, and if $\mathrm{Dom}(\Gamma'\Gamma_1) \supseteq \mathrm{Dom}(\Gamma_0)$, then $V_1; \Gamma_3\Gamma_2 \vdash_{\mathrm{styp}} s : \Gamma'\Gamma_1\Gamma_2$. So we can use SS6.8 to derive $V_1; \Gamma_3\Gamma_2 \vdash_{\mathrm{styp}} s : \Gamma_1$. Furthermore, if $\mathrm{Dom}(\Gamma_1) \supseteq \mathrm{Dom}(\Gamma_0)$, then $\mathrm{Dom}(\Gamma'\Gamma_1) \supseteq \mathrm{Dom}(\Gamma_0)$. So in this case, we can derive $V_1; \Gamma_3\Gamma_2 \vdash_{\mathrm{styp}} s : \Gamma_1\Gamma_2$ if $\Gamma_1\Gamma_2 \vdash_{\mathrm{wf}} \Gamma_1\Gamma_2$. We argued above that this result holds.

10. By inspection of the assumed derivation

11. By inspection of the assumed derivation

Lemma D.3 (Abstract-Ordering Antisymmetry).

1. *If* $\Gamma \vdash \textsc{none} \leq r$, *then* r *is* NONE.

2. *If* $\Gamma \vdash \textsc{all*} \leq r$, *then* r *is* ALL* *or* NONE.

3. *If* $\Gamma \vdash \textsc{all@} \leq r$, *then* r *is* ALL@, ALL*, *or* NONE.

4. *If* $\Gamma \vdash 0 \leq r$, *then* r *is* 0, ALL∗, *or* NONE.

5. *If* $\Gamma \vdash \&x \leq r$, *then* r *is* $\&x$, ALL@, ALL∗, *or* NONE. *If* r *is* ALL@ *or* ALL∗, *then* $\Gamma(x) = \tau, \text{ESC}, r'$ *for some* τ *and* r'.

6. *If* $\Gamma \vdash r_1 \leq r_2$, *then* r_1 *is* r_2 *or we* <u>*cannot*</u> *derive* $\Gamma' \vdash r_2 \leq r_1$.

7. *If* $\vdash k_1 \leq k_2$ *then* k_1 *is* k_2 *or we* <u>*cannot*</u> *derive* $\vdash k_2 \leq k_1$.

8. *If* $\Gamma \vdash \ell_1 \leq \ell_2$ *then* ℓ_1 *is* ℓ_2 *or we* <u>*cannot*</u> *derive* $\Gamma' \vdash \ell_2 \leq \ell_1$.

9. *If* $\Gamma \vdash \Gamma_1 \leq \Gamma_2$, *then* Γ_1 *is* Γ_2 *or we* <u>*cannot*</u> *derive* $\Gamma' \vdash \Gamma_2 \leq \Gamma_1$.

Proof: We prove each of the first five lemmas by induction on the assumed derivation. For the transitive case, we invoke the induction hypothesis twice and appeal to earlier lemmas.

For the sixth lemma, we proceed by cases on r_1. If r_1 is $\&x$ for some x, then the fifth lemma ensures four cases, which are trivial (r_2 is r_1) or handled by the first three lemmas. If r_1 is 0 (or ALL@), then the fourth lemma (or third lemma) ensures three cases, which are trivial or handled by the first two lemmas. If r_1 is ALL∗, then the second lemma ensures two cases, which are trivial or handled by the first lemma. If r_1 is NONE, the first lemma ensures there is one trivial case.

We prove the seventh lemma by cases on k_1.

We prove the eighth lemma by cases on ℓ_1.

We prove the ninth lemma by induction on the size of Γ_1, using the sixth and seventh lemmas.

Lemma D.4 (Abstract-Ordering Inversion).

If $\Gamma_0 \vdash \Gamma_1 \leq \Gamma_2$, *then for all* $x \in \text{Dom}(\Gamma_1)$, *there are one and only one* τ, k_1, k_2, r_1, *and* r_2 *such that* $\Gamma_1(x) = \tau, k_1, r_1$, $\Gamma_2(x) = \tau, k_2, r_2$, $\vdash k_1 \leq k_2$, *and* $\Gamma_0 \vdash r_1 \leq r_2$.

Proof: By induction on the assumed derivation

Lemma D.5 (Abstract-Ordering Transitivity).

1. *If* $\Gamma \vdash \ell_1 \leq \ell_2$ *and* $\Gamma \vdash \ell_2 \leq \ell_3$, *then* $\Gamma \vdash \ell_1 \leq \ell_3$.

2. *If* $\Gamma_1 \vdash \Gamma_0 \leq \Gamma_1$ *and* $\Gamma_0 \vdash \ell_1 \leq \ell_2$, *then* $\Gamma_1 \vdash \ell_1 \leq \ell_2$.

3. *If* $\vdash k_1 \leq k_2$ *and* $\vdash k_2 \leq k_3$, *then* $\vdash k_1 \leq k_3$.

4. *If* $\Gamma_1 \vdash \Gamma_0 \leq \Gamma_1$ *and* $\Gamma_0 \vdash r_1 \leq r_2$, *then* $\Gamma_1 \vdash r_1 \leq r_2$.

5. *If* $\Gamma_1 \vdash \Gamma_0 \leq \Gamma_1$ *and* $\Gamma_2 \vdash \Gamma_1 \leq \Gamma_2$, *then* $\Gamma_2 \vdash \Gamma_0 \leq \Gamma_2$.

Proof:

1. By inspection of the rules, ℓ_2 is ℓ_1 or ℓ_3.

2. By inspection of the rules, it suffices to show that if $\Gamma_0(x) = \tau, \text{ESC}, r$ for some x, τ, and r, then $\Gamma_1(x) = \tau, \text{ESC}, r'$ for some r'. The Abstract-Ordering Inversion Lemma ensures this result.

3. By inspection of the rules, k_2 is k_1 or k_3.

4. The proof is by induction on the derivation of $\Gamma_0 \vdash r_1 \leq r_2$. Every case is immediate or by induction except when the derivation is $\Gamma, x{:}\tau, \text{ESC}, r \vdash \&x \leq \text{ALL@}$ (so $\Gamma_0 = \Gamma, x{:}\tau, \text{ESC}, r$). In this case, the Abstract-Ordering Inversion Lemma ensures $\Gamma_1 = \Gamma', x{:}\tau, \text{ESC}, r'$ for some Γ' and r', so we can use the same rule to derive $\Gamma_1 \vdash \&x \leq \text{ALL@}$.

5. We prove the stronger result that if $\Gamma_1 \vdash \Gamma'_0 \leq \Gamma'_1$, $\Gamma_2 \vdash \Gamma'_1 \leq \Gamma'_2$, and $\Gamma_2 \vdash \Gamma_1 \leq \Gamma_2$, then $\Gamma_2 \vdash \Gamma'_0 \leq \Gamma'_2$. The proof is by induction on the sizes of Γ'_0, Γ'_1, and Γ'_2, which Typing Well-Formedness Lemma 2 ensures are the same. If $\Gamma'_0 = \cdot$, the result is trivial. For larger Γ'_0, we know Γ'_0, Γ'_1, and Γ'_2 have the form $\Gamma''_0, x{:}\tau, k_0, r_0$, $\Gamma''_1, x{:}\tau, k_1, r_1$, and $\Gamma''_2, x{:}\tau, k_2, r_2$, respectively. Inversion ensures $\Gamma_1 \vdash \Gamma''_0 \leq \Gamma''_1$, $\Gamma_2 \vdash \Gamma''_1 \leq \Gamma''_2$, $\vdash k_0 \leq k_1$, $\vdash k_1 \leq k_2$, $\Gamma_1 \vdash r_0 \leq r_1$ and $\Gamma_2 \vdash r_1 \leq r_2$. So induction ensures $\underline{\Gamma_2 \vdash \Gamma''_0 \leq \Gamma''_2}$. Abstract-Ordering Transitivity Lemma 3 ensures $\underline{\vdash k_0 \leq k_2}$. Because $\Gamma_2 \vdash \Gamma_1 \leq \Gamma_2$ and $\Gamma_1 \vdash r_0 \leq r_1$, Abstract-Ordering Transitivity Lemma 4 ensures $\Gamma_2 \vdash r_0 \leq r_1$, so with $\Gamma_2 \vdash r_1 \leq r_2$, we can derive $\underline{\Gamma_2 \vdash r_0 \leq r_2}$. From the underlined results, we can derive $\Gamma_2 \vdash \Gamma'_0 \leq \Gamma'_2$.

Lemma D.6 (Values Effectless).

1. *If* $\Gamma_0 \vdash_{\text{ltyp}} x : \tau, \ell, \Gamma_1$ *and* $\Gamma_0 \vdash_{\text{wf}} \Gamma_0$, *then* $\Gamma_1 \vdash \Gamma_0 \leq \Gamma_1$, $\Gamma_0(x) = \tau, k, r$ *(for some* k *and* r*), and* $\Gamma_1 \vdash x \leq \ell$.

2. *If* $\Gamma_0 \vdash_{\text{rtyp}} v : \tau, r, \Gamma_1$ *and* $\Gamma_0 \vdash_{\text{wf}} \Gamma_0$, *then* $\Gamma_1 \vdash \Gamma_0 \leq \Gamma_1$. *Furthermore, if* $v = \&x$, *then there exists* τ', k, *and* r' *such that* $\Gamma_0(x) = \tau', k, r'$, τ *is* $\tau'*$ *or* $\tau'@$, *and* $\Gamma_1 \vdash \&x \leq r$.

3. *If* $V; \Gamma_0 \vdash_{\text{styp}} v : \Gamma_1$ *and* $\Gamma_0 \vdash_{\text{wf}} \Gamma_0$, *then there exists a* Γ_2 *such that* $\Gamma_1\Gamma_2 \vdash \Gamma_0 \leq \Gamma_1\Gamma_2$ *and* $\Gamma_1\Gamma_2 \vdash_{\text{wf}} \Gamma_2$.

Proof:

1. The proof is by induction on the assumed typing derivation, which must end with SL6.1, SL6.3, or SL6.4. Case SL6.1 follows from $\Gamma \vdash x \leq x$ and $\Gamma \vdash \Gamma \leq \Gamma$ for any Γ. Case SL6.3 follows from induction and Abstract-Ordering Transitivity Lemmas 2 and 5. Case SL6.4 follows from induction and Abstract-Ordering Transitivity Lemma 1.

2. The proof is by induction on the assumed typing derivation, which must end with SR6.1–4, SR6.7A–B, or SR6.11–13. Cases SR6.1–4 follow from $\Gamma \vdash \Gamma \leq \Gamma$ for any Γ. Cases SR6.7A–B follow from the previous lemma. (Case SR6.7B requires inverting the derivation of $\Gamma_1 \vdash x \leq ?$ to derive $\Gamma_1 \vdash \&x \leq \textsc{all@}$.) Cases SR6.11 follows from induction. Case SR6.12 follows from induction and Abstract-Ordering Transitivity Lemmas 4 and 5. and SR6.13 follow from induction and the transitivity rule for abstract-rvalue ordering.

3. The proof is by induction on the assumed typing derivation, which must end with SS6.1, or SS6.7–8. Case SS6.1 follows from the previous lemma, letting $\Gamma_2 = \cdot$. For case SS6.8, inversion ensures $V; \Gamma_0 \vdash_{\text{styp}} v : \Gamma'\Gamma_1$ for some Γ'. So induction ensures there exists a Γ'_2 such that $\Gamma'\Gamma'_2\Gamma_1 \vdash \Gamma_0 \leq \Gamma'\Gamma'_2\Gamma_1$ and $\Gamma'\Gamma'_2\Gamma_1 \vdash_{\text{wf}} \Gamma'_2$. Typing Well-Formedness Lemma 5 ensures $\Gamma'\Gamma_1 \vdash_{\text{wf}} \Gamma'\Gamma_1$. So Weakening Lemma 3 ensures $\Gamma'\Gamma'_2\Gamma_1 \vdash_{\text{wf}} \Gamma'_2\Gamma'\Gamma_1$. So letting $\Gamma_2 = \Gamma'\Gamma'_2$ suffices.

   For case SS6.7, inversion ensures $V; \Gamma_0 \vdash_{\text{styp}} v : \Gamma'$ and $\Gamma_1 \vdash \Gamma' \leq \Gamma_1$ for some Γ'. By induction there exists a Γ'_2 such that $\Gamma'\Gamma'_2 \vdash \Gamma_0 \leq \Gamma'\Gamma'_2$ and $\Gamma'\Gamma'_2 \vdash_{\text{wf}} \Gamma'_2$. Because $\Gamma_1 \vdash \Gamma' \leq \Gamma_1$, Weakening Lemma 6 ensures $\Gamma_1\Gamma'_2 \vdash \Gamma'\Gamma'_2 \leq \Gamma_1\Gamma'_2$. So Abstract-Ordering Transitivity Lemma 5 ensures $\Gamma_1\Gamma'_2 \vdash \Gamma_0 \leq \Gamma_1\Gamma'_2$. Because $\Gamma_1 \vdash \Gamma' \leq \Gamma_1$, Typing Well-Formedness Lemma 2 ensures $\text{Dom}(\Gamma_1) = \text{Dom}(\Gamma')$, so $\Gamma'\Gamma'_2 \vdash_{\text{wf}} \Gamma'_2$ and Typing Well-Formedness Lemma 1 ensure $\Gamma_1\Gamma'_2 \vdash_{\text{wf}} \Gamma'_2$. So letting $\Gamma_2 = \Gamma'_2$ suffices.

Lemma D.7 (Heap Subsumption). *Suppose* $\Gamma \vdash_{\text{wf}} \Gamma$, $\Gamma' \vdash \Gamma \leq \Gamma'$, *and* $\Gamma' \vdash_{\text{wf}} \Gamma'$.

1. *If* $\Gamma \vdash_{\text{ltyp}} x : \tau, \ell, \Gamma$ *and* $\Gamma' \vdash \ell \leq \ell'$, *then* $\Gamma' \vdash_{\text{ltyp}} x : \tau, \ell', \Gamma'$.

2. *If* $\Gamma \vdash_{\text{rtyp}} v : \tau, r, \Gamma$ *and* $\Gamma' \vdash r \leq r'$, *then* $\Gamma' \vdash_{\text{rtyp}} v : \tau, r', \Gamma'$.

3. *If* $\Gamma = \Gamma_0\Gamma_1$, $\Gamma' = \Gamma'_0\Gamma'_1$, $\text{Dom}(\Gamma_1) = \text{Dom}(\Gamma'_1)$, *and* $\Gamma \vdash_{\text{htyp}} H : \Gamma_1$, *then* $\Gamma' \vdash_{\text{htyp}} H : \Gamma'_1$.

Proof:

1. The proof is by induction on the derivation of $\Gamma \vdash_{\text{ltyp}} x : \tau, \ell, \Gamma$, which must end with SL6.1, SL6.3, or SL6.4. For case SL6.1, the Abstract-Ordering

Inversion Lemma ensures Γ and Γ' give the same type τ to x, so SL6.1 lets us derive $\Gamma' \vdash_{\text{ltyp}} x : \tau, \ell, \Gamma'$. Then SL6.4 lets us derive $\Gamma' \vdash_{\text{ltyp}} x : \tau, \ell', \Gamma'$

For case SL6.3, inversion ensures $\Gamma \vdash_{\text{ltyp}} x : \tau, \ell, \Gamma_1$ and $\Gamma \vdash \Gamma_1 \leq \Gamma$ for some Γ. So Values Effectless Lemma 1 ensures $\Gamma_1 \vdash \Gamma \leq \Gamma_1$. So Abstract-Ordering Antisymmetry Lemma 9 ensures $\Gamma = \Gamma_1$ and the result follows from induction (because we have a shorter derivation of $\Gamma \vdash_{\text{ltyp}} x : \tau, \ell, \Gamma$).

For case SL6.4, inversion ensures $\Gamma \vdash_{\text{ltyp}} x : \tau, \ell_1, \Gamma$ and $\Gamma \vdash \ell_1 \leq \ell$ for some ℓ_1. So Abstract-Ordering Transitivity Lemma 2 ensures $\Gamma' \vdash \ell_1 \leq \ell$. So Abstract-Ordering Transitivity Lemma 1 ensures $\Gamma' \vdash \ell_1 \leq \ell'$. By induction and $\Gamma' \vdash \ell_1 \leq \ell_1$, we know $\Gamma' \vdash_{\text{ltyp}} x : \tau, \ell_1, \Gamma'$. So we can use SL6.4 to derive $\Gamma' \vdash_{\text{ltyp}} x : \tau, \ell', \Gamma'$.

2. The proof is by induction on the assumed typing derivation, which must end with SR6.1–4, SR6.7A–B, or SR6.11–13. For cases SR6.1–4, we can use the same rule to derive $\Gamma' \vdash_{\text{rtyp}} v : \tau, r, \Gamma'$. Then SR6.13 lets us derive $\Gamma' \vdash_{\text{rtyp}} v : \tau, r', \Gamma'$. For cases SR6.7A–B, $v = \&x$ and inversion ensures $\Gamma \vdash_{\text{ltyp}} x : \tau, \ell, \Gamma$ for some ℓ. Because $\Gamma' \vdash \ell \leq \ell$, the previous lemma ensures $\Gamma' \vdash_{\text{ltyp}} x : \tau, \ell, \Gamma'$. Then SR6.7A or SR6.7B lets us derive $\Gamma' \vdash_{\text{rtyp}} \&x : \tau, r, \Gamma'$. Then SR6.13 lets us derive $\Gamma' \vdash_{\text{rtyp}} v : \tau, r', \Gamma'$. Case SR6.11 follows from induction.

   Cases SR6.12 and SR6.13 are analogous to cases SL6.3 and SL6.4 in the previous proof. Case SR6.12 uses Values Effectless Lemma 2 instead of 1. Case SR6.13 uses Abstract-Ordering Transitivity Lemma 4 instead of 2, and the transitivity rule for abstract-rvalue ordering instead of Abstract-Ordering Transitivity Lemma 1.

3. The proof is by induction on the derivation of $\Gamma \vdash_{\text{htyp}} H : \Gamma_1$. If $H = \cdot$, then $\Gamma_1 = \Gamma_1' = \cdot$ and the result is immediate. Else inversion ensures there exists Γ_2, k, r, Γ_2', k', r', τ, H', and v where $\Gamma_1 = \Gamma_2, x{:}\tau, k, r$, $\Gamma_1' = \Gamma_2', x{:}\tau, k', r'$, $H = H', x \mapsto v$, and the typing derivation ends as follows:

$$\frac{\Gamma \vdash_{\text{htyp}} H' : \Gamma_2 \quad \Gamma \vdash_{\text{rtyp}} v : \tau, r, \Gamma}{\Gamma \vdash_{\text{htyp}} H', x \mapsto v : \Gamma_2, x{:}\tau, k, r}$$

   Therefore, the induction hypothesis ensures $\Gamma' \vdash_{\text{htyp}} H' : \Gamma_2'$, so it suffices to show $\Gamma' \vdash_{\text{rtyp}} v : \tau, r', \Gamma'$. The Abstract-Ordering Inversion Lemma ensures $\Gamma' \vdash r \leq r'$. So the result follows from Heap Subsumption Lemma 2.

Lemma D.8 (Value Elimination). *If* $\Gamma_0 \vdash_{\text{htyp}} H : \Gamma_0$, $\Gamma_0 \vdash_{\text{wf}} \Gamma_0$, $V'; \Gamma_0 \vdash_{\text{styp}} v : \Gamma_3$, $V'; \Gamma_3 \vdash_{\text{styp}} s : \Gamma_1$, $\vdash_{\text{V}} s : V''$, $V'' \subseteq V'$, $V' \cap \text{Dom}(H) = \emptyset$, *and* $V \supseteq \text{Dom}(H) \cup V'$, *then* $\vdash_{\text{prog}} V; H; s : \Gamma_1$.

Proof: Because $V';\Gamma_0 \vdash_{\text{styp}} v : \Gamma_3$, Typing Well-Formedness Lemma 5 ensures $\Gamma_3 \vdash_{\text{wf}} \Gamma_3$. So Values Effectless Lemma 3 ensures there exists a Γ_2 such that $\Gamma_3\Gamma_2 \vdash \Gamma_0 \leq \Gamma_3\Gamma_2$ and $\underline{\Gamma_3\Gamma_2 \vdash_{\text{wf}} \Gamma_3\Gamma_2}$. Typing Well-Formedness Lemma 2 ensures $\text{Dom}(\Gamma_3\Gamma_2) = \text{Dom}(\Gamma_0)$. So Heap Subsumption Lemma 3 ensures $\underline{\Gamma_3\Gamma_2 \vdash_{\text{htyp}} H : \Gamma_3\Gamma_2}$. Given the lemma's assumptions and the underlined results, we can derive $\vdash_{\text{prog}} V; H; s : \Gamma_1$ if $\underline{V';\Gamma_3\Gamma_2 \vdash_{\text{styp}} s : \Gamma_1}$.

We show that Weakening Lemma 9 ensures $V';\Gamma_3\Gamma_2 \vdash_{\text{styp}} s : \Gamma_1$ by instantiating this lemma with Γ_3 for Γ_0, Γ_2 for Γ_2, V' for V_0, Γ_3 for Γ_3, V' for V_1, and V'' for V_2. (The key is that the lemma distinguishes its Γ_3 from Γ_0 and V_0 from V_1 for its inductive proof, but we use Γ_3 and V' for both.) Because $\Gamma_3\Gamma_2 \vdash_{\text{wf}} \Gamma_3\Gamma_2$, inversion ensures $\Gamma_3\Gamma_2 \vdash_{\text{wf}} \Gamma_2$. We show $V' \cap \text{Dom}(\Gamma_3\Gamma_2) = \emptyset$ as follows: A trivial induction on $\Gamma_0 \vdash_{\text{htyp}} H : \Gamma_0$ shows $\text{Dom}(\Gamma_0) = \text{Dom}(H)$. We showed above that $\text{Dom}(\Gamma_3\Gamma_2) = \text{Dom}(\Gamma_0)$. So $\text{Dom}(\Gamma_3\Gamma_2) = \text{Dom}(H)$. By assumption $V' \cap \text{Dom}(H) = \emptyset$. So $V' \cap \text{Dom}(\Gamma_3\Gamma_2) = \emptyset$. Trivially, $\text{Dom}(\Gamma_3) \subseteq \text{Dom}(\Gamma_3) \subseteq \text{Dom}(\Gamma_3) \cup V$. By assumption, $V'' \subseteq V'$, so $V'' \subseteq V' \subseteq V'$. We showed above that $\Gamma_3 \vdash_{\text{wf}} \Gamma_3$. By assumption $V';\Gamma_3 \vdash_{\text{styp}} s : \Gamma_1$ and $\vdash_{\text{V}} s : V''$. So the lemma applies.

Lemma D.9 (Canonical Forms). *Suppose* $\Gamma \vdash_{\text{wf}} \Gamma$.

1. *If* $\Gamma \vdash_{\text{rtyp}} v : \tau, 0, \Gamma'$, *then* v *is* 0.
2. *If* $\Gamma \vdash_{\text{rtyp}} v : \text{int}, r, \Gamma'$, *then* r *is not* $\&x$.
3. *If* $\Gamma \vdash_{\text{rtyp}} v : \text{int}, \text{ALL@}, \Gamma'$, *then* v *is some* $i \neq 0$.
4. *If* $\Gamma \vdash_{\text{rtyp}} v : \text{int}, \text{ALL*}, \Gamma'$, *then* v *is some* i.
5. *If* $\Gamma \vdash_{\text{rtyp}} v : \tau, \&x, \Gamma'$ *and* $\tau \neq \text{int}$, *then* v *is* $\&x$.
6. *If* $\Gamma \vdash_{\text{rtyp}} v : \tau, \text{ALL@}, \Gamma'$ *and* $\tau \neq \text{int}$, *then* v *is* $\&x$ *for some* x.
7. *If* $\Gamma \vdash_{\text{rtyp}} v : \tau, \text{ALL*}, \Gamma'$ *and* $\tau \neq \text{int}$, *then* v *is* 0 *or* $\&x$ *for some* x.
8. *If* $\Gamma \vdash_{\text{rtyp}} v : \tau*, r, \Gamma'$, $\Gamma \vdash_{\text{wf}} \tau@, k, r$, *and* $r \neq \text{NONE}$, *then* $\Gamma \vdash_{\text{rtyp}} v : \tau@, r, \Gamma'$.
9. *If* $\Gamma \vdash_{\text{rtyp}} v : \tau, r, \Gamma'$ *and* $r \neq \text{NONE}$, *then* $v \neq \mathsf{junk}$.

Proof:

1. The proof is by induction on the assumed derivation, which must end with SR6.2–3 or SR6.11–13. Cases SR6.2 and SR6.3 are trivial. Cases SR6.11–12 follows from induction. Case SR6.13 follows from induction because the Abstract-Ordering Antisymmetry Lemmas ensure for any Γ and r, $\Gamma \vdash r \leq 0$ only if r is 0. (In fact, case SR6.11 is impossible.)

2. The proof is by induction on the assumed derivation, which must end with SR6.3–4 or SR6.12–13. Cases SR6.3 and SR6.4 are trivial. Case SR6.12 follows from induction. Case SR6.13 follows from induction because the Abstract-Ordering Antisymmetry Lemmas ensure for any Γ, r_1, r_2, x, and y, if $\Gamma \vdash r_1 \leq r_2$ and r_1 is not $\&x$, then r_2 is not $\&y$.

3. The proof is by induction on the assumed derivation, which must end with SR6.4 or SR6.12–13. Case SR6.4 is trivial. Case SR6.12 follows from induction. Case SR6.13 follows from induction because the Abstract-Ordering Antisymmetry Lemmas ensure for any Γ and r, $\Gamma \vdash r \leq$ ALL@ only if r is ALL@ or $\&x$. Canonical Forms Lemma 2 eliminates the latter possibility.

4. The proof is by induction on the assumed derivation, which must end with SR6.12–13. Case SR6.12 follows from induction. For case SR6.13, the Abstract-Ordering Antisymmetry Lemmas ensure for any Γ and r, $\Gamma \vdash r \leq$ ALL* only if r is not NONE. From this fact, Canonical Forms Lemma 2, and inversion, we know $\Gamma \vdash_{\text{rtyp}} v : \text{int}, r', \Gamma'$ where r' is one of ALL*, ALL@, or 0. If r' is ALL*, the result follows from induction. If r' is ALL@, the result follows from Canonical Forms Lemma 3. If r' is 0, the result follows from Canonical Forms Lemma 1.

5. The proof is by induction on the assumed derivation, which must end in SR6.7A, or SR6.11–13. For case SR6.7A, inversion ensures v is $\&y$ for some y. So Values Effectless Lemma 2 ensures $\Gamma' \vdash \&y \leq \&x$. So Abstract-Ordering Antisymmetry Lemma 5 ensures y is x. Cases SR6.11–12 follow from induction. Case SR6.13 follows from induction because the Abstract-Ordering Antisymmetry Lemmas ensure for any Γ and r, $\Gamma \vdash r \leq \&x$ only if r is $\&x$.

6. The proof is by induction on the assumed derivation, which must end with SR6.7B or SR6.11–13. Case SR6.7B is trivial. Cases SR6.11–12 follow from induction. For case SR6.13, the Abstract-Ordering Antisymmetry Lemmas ensure for any Γ and r, $\Gamma \vdash r \leq$ ALL@ only if r is ALL@ or $\&x$ for some x. From this fact and inversion, we know $\Gamma \vdash_{\text{rtyp}} v : \tau, r', \Gamma'$ where r' is ALL@ or $\&x$ for some x. If r' is ALL@, the result follows from induction. If r' is some $\&x$, Canonical Forms Lemma 5 ensures v is $\&x$.

7. The proof is by induction on the assumed derivation, which must end with SR6.11–SR6.13. Cases SR6.11–12 follow from induction. For case SR6.13, the Abstract-Ordering Antisymmetry Lemmas ensure for any Γ and r, $\Gamma \vdash r \leq$ ALL* implies r is not NONE. From this fact and inversion, we know

$\Gamma \vdash_{\text{rtyp}} v : \tau, r', \Gamma'$ where r' is ALL$*$, 0, ALL@ or $\&x$ for some x. If r' is ALL$*$, then the result follows from induction. If r' is ALL@, the result follows from Canonical Forms Lemma 6. If r' is 0, the result follows from Canonical Forms Lemma 1. If r' is $\&x$, Canonical Forms Lemma 5 ensures v is some $\&x$.

8. The proof is by induction on the assumed typing derivation, which must end with SR6.2 or SR6.11–13. Case SR6.2 holds vacuously because $\Gamma \not\vdash_{\text{wf}} \tau@, k, 0$. Case SR6.11 holds by inversion. Case SR6.12 follows from induction. For case SR6.13, inversion ensures $\Gamma \vdash_{\text{rtyp}} v : \tau@, r', \Gamma'$ and $\Gamma' \vdash r' \leq r$. Because $\Gamma \vdash_{\text{wf}} \tau@, k, r$ and $r \neq$ NONE, r is ALL@ or $\&x$ for some $x \in \text{Dom}(\Gamma)$. So the Abstract-Ordering Antisymmetry Lemmas ensure r' is ALL@ or $\&y$ for some $y \in \text{Dom}(\Gamma)$. In either case, $\Gamma \vdash_{\text{wf}} \tau@, k', r'$ for some k' and $r' \neq$ NONE. So induction ensures $\Gamma \vdash_{\text{rtyp}} v : \tau*, r', \Gamma'$ and SR6.13 lets us derive $\Gamma \vdash_{\text{rtyp}} v : \tau*, r, \Gamma'$.

9. This lemma is a corollary of Canonical Forms Lemmas 1 and 3–7 because $r \neq$ NONE ensures one of these lemmas applies (using Canonical Forms Lemma 2 when $r = \&x$ for some x).

Lemma D.10 (Subtyping Preservation).

Suppose $\Gamma \vdash_{\text{wf}} \Gamma$ *and* $\tau = \tau_0*$ *or* $\tau = \tau_0@$.

1. *If* $\Gamma \vdash_{\text{rtyp}} \&x : \tau, \&x, \Gamma'$, *then* $\Gamma \vdash_{\text{ltyp}} x : \tau_0, x, \Gamma'$.

   If $\Gamma \vdash_{\text{rtyp}} \&x : \tau, \text{ALL@}, \Gamma'$, *then* $\Gamma \vdash_{\text{ltyp}} x : \tau_0, ?, \Gamma'$.

2. *If* $\Gamma \vdash_{\text{ltyp}} x : \tau_0, x, \Gamma'$, *then* $\Gamma'(x) = \tau_0, k, r$ *for some* k *and* r, *and* $\Gamma \vdash_{\text{rtyp}} x : \tau_0, r, \Gamma'$.

   If $\Gamma \vdash_{\text{ltyp}} x : \tau_0, ?, \Gamma'$, *then there exists an* r *such that* $\Gamma' \vdash_{\text{wf}} \tau_0, \text{ESC}, r$ *and* $\Gamma \vdash_{\text{rtyp}} x : \tau_0, r, \Gamma'$.

3. *If* $\Gamma \vdash_{\text{rtyp}} \&x : \tau, \&x, \Gamma'$, *then* $\Gamma'(x) = \tau_0, k, r$ *for some* k *and* r, *and* $\Gamma \vdash_{\text{rtyp}} x : \tau_0, r, \Gamma'$.

   If $\Gamma \vdash_{\text{rtyp}} \&x : \tau, \text{ALL@}, \Gamma'$, *then there exists an* r *such that* $\Gamma' \vdash_{\text{wf}} \tau_0, \text{ESC}, r$ *and* $\Gamma \vdash_{\text{rtyp}} x : \tau_0, r, \Gamma'$.

Proof:

1. The proof is by induction on the assumed derivations, which must end with SR6.7A–B or SR6.11–13. Cases SR6.7A–B follow from inversion. Case 6.11 follows from induction. For case SR6.12, inversion ensures $\Gamma \vdash_{\text{rtyp}} \&x : \tau, r, \Gamma''$,

$\Gamma' \vdash \Gamma'' \leq \Gamma'$, and $\Gamma' \vdash_{\text{wf}} \Gamma'$ for some Γ''. So induction ensures $\Gamma \vdash_{\text{ltyp}} x : \tau, \ell, \Gamma''$ for the appropriate ℓ. So SL6.3 lets us derive $\Gamma \vdash_{\text{ltyp}} x : \tau, \ell, \Gamma'$.

For case SR6.13, inversion ensures $\Gamma \vdash_{\text{rtyp}} e : \tau, r', \Gamma'$ and $\Gamma' \vdash r' \leq r$. If r is $\&x$, the Abstract-Ordering Antisymmetry Lemmas ensure r' is $\&x$, so the result follows from induction. If r is ALL@, the Abstract-Ordering Antisymmetry Lemmas ensure r' is ALL@ or $\&y$ for some y. If r' is ALL@, the result follows from induction. Otherwise, $\Gamma \vdash_{\text{rtyp}} e : \tau, r', \Gamma'$ and Canonical Forms Lemma 5 ensure $y = x$. So induction ensures $\Gamma \vdash_{\text{ltyp}} x : \tau_0, x, \Gamma'$. Furthermore, $\Gamma' \vdash \&x \leq$ ALL@ and Abstract-Ordering Antisymmetry Lemma 5 ensure $\Gamma'(x) = \tau_0, \text{ESC}, r$ for some r. So we can derive $\Gamma' \vdash x \leq ?$. So SL6.4 lets us derive $\Gamma \vdash_{\text{ltyp}} x : \tau_0, ?, \Gamma'$.

2. The proof is by induction on the assumed derivations, which must end with SL6.1 or SL6.3–4. For case SL6.1, the first statement follows by using SR6.6 to derive the result. The second statement holds vacuously.

   For case SL6.3, inversion ensures $\Gamma \vdash_{\text{ltyp}} x : \tau, \ell, \Gamma''$, $\Gamma' \vdash \Gamma'' \leq \Gamma'$, and $\Gamma' \vdash_{\text{wf}} \Gamma'$ for some Γ'' and the appropriate ℓ. So by induction, the results hold using Γ'' in place of Γ'. So SR6.12 lets us derive the typing results we need. For the other results, $\Gamma' \vdash \Gamma'' \leq \Gamma'$, the Abstract-Ordering Inversion Lemma, and $\Gamma''(x) = \tau_0, k, r$ for some k and r ensure $\Gamma'(x) = \tau_0, k', r'$ for some k' and r'. Furthermore, $\Gamma'' \vdash_{\text{wf}} \tau_0, \text{ESC}, r$ ensures $\Gamma' \vdash_{\text{wf}} \tau_0, \text{ESC}, r'$ because the typing context is irrelevant when k is ESC.

   For case SL6.4, inversion ensures $\Gamma \vdash_{\text{ltyp}} x : \tau, \ell', \Gamma'$ and $\Gamma' \vdash \ell' \leq \ell$. If ℓ is x, then inspection of $\Gamma' \vdash \ell' \leq \ell$ ensures ℓ' is x, so the result follows from induction. If ℓ is ?, then inspection of $\Gamma' \vdash \ell' \leq \ell$ ensures ℓ' is x or ?. (Values Effectless Lemma 1 ensures ℓ' is not some $y \neq x$.) If ℓ' is ?, the result follows from induction. Otherwise, induction ensures $\Gamma \vdash_{\text{rtyp}} x : \tau_0, r, \Gamma'$ where $\Gamma'(x) = \tau_0, k, r$. Inverting $\Gamma' \vdash x \leq ?$ ensures k is ESC. The Typing Well-Formedness Lemma ensures $\Gamma' \vdash_{\text{wf}} \Gamma'$, so $\Gamma' \vdash_{\text{wf}} \tau_0, \text{ESC}, r$, as required.

3. This lemma is a corollary of the previous two lemmas.

Lemma D.11 (Assignment Preservation).

Suppose $\Gamma_0 = \Gamma, x{:}\tau, k, r_0$, $\Gamma_1 = \Gamma, x{:}\tau, k, r_1$, $\Gamma_0 \vdash_{\text{wf}} \Gamma_0$, *and* $\Gamma_1 \vdash_{\text{wf}} \Gamma_1$.

1. *If* $\Gamma_0 \vdash \ell \leq \ell'$, *then* $\Gamma_1 \vdash \ell \leq \ell'$.
2. *If* $\Gamma_0 \vdash r \leq r'$, *then* $\Gamma_1 \vdash r \leq r'$.
3. *If* $\Gamma_0 \vdash_{\text{ltyp}} y : \tau, \ell, \Gamma_0$, *then* $\Gamma_1 \vdash_{\text{ltyp}} y : \tau, \ell, \Gamma_1$.

4. *If* $\Gamma_0 \vdash_{\text{rtyp}} v : \tau, r, \Gamma_0$, *then* $\Gamma_1 \vdash_{\text{rtyp}} v : \tau, r, \Gamma_1$.

5. *If* $\Gamma = \Gamma_A \Gamma_B$ *and* $\Gamma_0 \vdash_{\text{htyp}} H : \Gamma_B$, *then* $\Gamma_1 \vdash_{\text{htyp}} H : \Gamma_B$.

Proof:

1. The proof is by cases on the assumed derivation. The abstract rvalues in the typing context are irrelevant.

2. The proof is by induction on the assumed derivation. The abstract rvalues in the typing context are irrelevant.

3. The proof is by induction on the assumed derivation, which must end with SL6.1, SL6.3, or SL6.4. Case SL6.1 is trivial (even if y is x). For case SL6.3, inversion ensures $\Gamma \vdash_{\text{ltyp}} x : \tau, \ell, \Gamma_1$ and $\Gamma \vdash \Gamma_1 \leq \Gamma$ for some Γ. So Values Effectless Lemma 1 ensures $\Gamma_1 \vdash \Gamma \leq \Gamma_1$. So Abstract-Ordering Antisymmetry Lemma 9 ensures $\Gamma = \Gamma_1$ and the result follows from induction (because we have a shorter derivation of $\Gamma \vdash_{\text{ltyp}} x : \tau, \ell, \Gamma$). Case SL6.3 follows from induction (changing Γ'). Case SL6.4 follows from induction and Assignment Preservation Lemma 1 (to use SL6.4 to derive the result).

4. The proof is by induction on the assumed derivation, which must end with SR6.1–4, SR6.7A–B, or SR6.11–13. Cases SR6.1–4 follow trivially. Cases SR6.7A–B follow from Assignment Preservation Lemma 3. Case SR6.11 follows from induction (using SR6.11 to derive the result). Case SR6.12 follows from an argument analogous to case SL6.3 in the previous proof. Case SR6.13 follows from induction and Assignment Preservation Lemma 2 (to use SR6.13 to derive the result).

5. The proof is by induction on the assumed derivation. If $H = \cdot$, the result is trivial. Else the result follows from induction and Assignment Preservation Lemma 4.

Lemma D.12 (Systematic Renaming). *If* $V_1; \Gamma_0 \vdash_{\text{styp}} s : \Gamma_1$, $\vdash_{\text{V}} s : V_2$, $V_2 \subseteq V_1 \subseteq V_0$, $\text{Dom}(M) = \text{Dom}(V_2)$, $V_0 \vdash_{\text{wf}} M$, *and* $\vdash_{\text{V}} \text{rename}(M, s) : V'$, *then* $V_1 \cup V'; \text{rename}(M, \Gamma_0) \vdash_{\text{styp}} \text{rename}(M, s) : \text{rename}(M, \Gamma_1)$ *and* $V_0 \cap V' = \emptyset$.

Proof: The proof is by induction on the assumed typing derivation using many (omitted) renaming lemmas for other judgments. As examples, we can prove $\Gamma_0 \vdash \Gamma_1 \leq \Gamma_2$ ensures $\text{rename}(M, \Gamma_0) \vdash \text{rename}(M, \Gamma_1) \leq \text{rename}(M, \Gamma_2)$ and $\Gamma_0 \vdash_{\text{rtyp}} e : \tau, r, \Gamma_1$ ensures $\text{rename}(M, \Gamma_0) \vdash_{\text{rtyp}} \text{rename}(M, e) : \tau, \text{rename}(M, r), \text{rename}(M, \Gamma_1)$. None of these lemmas are interesting.

Lemma D.13 (Useless Renaming). *If* $\Gamma \vdash_{\text{wf}} \Gamma$ *and* $\text{Dom}(\Gamma) \cap \text{Dom}(M) = \emptyset$, *then* $\text{rename}(M, \Gamma) = \Gamma$.

Proof: By induction on the derivation of $\Gamma \vdash_{\text{wf}} \Gamma$

Lemma D.14 (Preservation). *Suppose* $\Gamma_0 \vdash_{\text{htyp}} H : \Gamma_0$ *and* $\Gamma_0 \vdash_{\text{wf}} \Gamma_0$.

1. *If* $\Gamma_0 \vdash_{\text{ltyp}} e : \tau, \ell, \Gamma_1$ *and* $H; e \xrightarrow{\text{l}} H'; e'$, *then there exists a* Γ_2 *with* $\text{Dom}(\Gamma_2) = \text{Dom}(\Gamma_0)$ *such that* $\Gamma_2 \vdash_{\text{wf}} \Gamma_2$, $\Gamma_2 \vdash_{\text{htyp}} H' : \Gamma_2$, *and* $\Gamma_2 \vdash_{\text{ltyp}} e' : \tau, \ell, \Gamma_1$. *Furthermore, if* $\Gamma_0 \vdash \Gamma_1 \leq \Gamma_0$, *then one such* Γ_2 *is* Γ_0.

   If $\Gamma_0 \vdash_{\text{rtyp}} e : \tau, r, \Gamma_1$ *and* $H; e \xrightarrow{\text{r}} H'; e'$, *then there exists a* Γ_2 *with* $\text{Dom}(\Gamma_2) = \text{Dom}(\Gamma_0)$ *such that* $\Gamma_2 \vdash_{\text{wf}} \Gamma_2$, $\Gamma_2 \vdash_{\text{htyp}} H' : \Gamma_2$ *and* $\Gamma_2 \vdash_{\text{rtyp}} e' : \tau, r, \Gamma_1$. *Furthermore, if* $\Gamma_0 \vdash \Gamma_1 \leq \Gamma_0$, *then one such* Γ_2 *is* Γ_0.

2. *If* $V; \Gamma_0 \vdash_{\text{tst}} e : \Gamma_1; \Gamma_2$ *and* $H; e \xrightarrow{\text{r}} H'; e'$, *then there exists a* Γ_3 *with* $\text{Dom}(\Gamma_3) = \text{Dom}(\Gamma_0)$ *such that* $\Gamma_3 \vdash_{\text{wf}} \Gamma_3$, $\Gamma_3 \vdash_{\text{htyp}} H' : \Gamma_3$, *and* $V; \Gamma_3 \vdash_{\text{tst}} e' : \Gamma_1; \Gamma_2$.

3. *Suppose* $\vdash_{\text{prog}} V_0; H; s : \Gamma_1$, $V_0; H; s \xrightarrow{\text{s}} V_1; H'; s'$, *and* $\vdash_{\text{v}} s : V_0''$. *Then* $\vdash_{\text{prog}} V_1; H'; s' : \Gamma_1$, $V_1 \supseteq V_0$, *and* $\text{Dom}(H') \subseteq \text{Dom}(H) \cup V_0''$. *Furthermore, if* $\vdash_{\text{v}} s' : V_1''$, *then* $V_1'' \cap V_0 \subseteq V_0''$.

Proof:

1. The proof is by simultaneous induction on the assumed typing derivations, proceeding by cases on the last rule used. Note that if $\Gamma_1 = \Gamma_0$, then a trivial induction ensures $\Gamma_0 \vdash \Gamma_1 \leq \Gamma_0$.

   - SR6.1–4: These cases are trivial because no rule applies.
   - SR6.5: Let $e = \star$ and $\Gamma_0 = \Gamma_1 = \Gamma$. Only rule DR6.4 applies, so $e' = i$ for some i and $H' = H$. Using SR6.3 or SR6.4 (depending on whether i is 0), we can derive $\Gamma \vdash_{\text{rtyp}} i : \text{int}, r, \Gamma$ for $r = \text{ALL@}$ or $r = 0$. In either case, we can derive $\Gamma \vdash r \leq \text{ALL}*$, so SR6.13 lets us derive $\Gamma \vdash_{\text{rtyp}} i : \text{int}, \text{ALL}*, \Gamma$.
   - SR6.6: Let $e = x$, $\Gamma_0 = \Gamma_1 = \Gamma$, and $\Gamma(x) = \tau, k, r$. Only rule DR6.1 applies, so $e' = H(x)$ and $H' = H$. Inverting $\Gamma \vdash_{\text{htyp}} H : \Gamma$ ensures $\Gamma \vdash_{\text{rtyp}} H(x) : \tau, r, \Gamma$, as required.
   - SR6.7A: Let $e = \&e_0$ and $\tau = \tau_0@$. Only rule DR6.7 applies, so $e' = \&e_0'$ where $H; e_0 \xrightarrow{\text{l}} H'; e_0'$. By inversion, $\Gamma_0 \vdash_{\text{ltyp}} e_0 : \tau_0, x, \Gamma_1$. So by induction, there exists a Γ_2 such that $\underline{\Gamma_2 \vdash_{\text{wf}} \Gamma_2}$, $\underline{\Gamma_2 \vdash_{\text{htyp}} H' : \Gamma_2}$, and $\Gamma_2 \vdash_{\text{ltyp}} e_0' : \tau_0, x, \Gamma_1$ (and if $\Gamma_0 \vdash \Gamma_1 \leq \Gamma_0$, then one such Γ_2 is Γ_0). So SR6.7A lets us derive $\underline{\Gamma_2 \vdash_{\text{rtyp}} \&e_0' : \tau, \&x, \Gamma_1}$ (and if $\Gamma_0 \vdash \Gamma_1 \leq \Gamma_0$, then $\Gamma_0 \vdash_{\text{rtyp}} \&e_0' : \tau, \&x, \Gamma_1$).

- SR6.7B: This case is analogous to the previous one, using SR6.7B in place of SR6.6A, ALL@ in place of $\&x$, and ? in place of x.
- SR6.8A: Let $e = *e_0$. By inversion, $\Gamma_0 \vdash_{\text{rtyp}} e_0 : \tau*, \&x, \Gamma_1$ and $\Gamma_1(x) = \tau, k, r$ for some k. Only rules DR6.3 and DR6.6 apply.

  For DR6.3, Canonical Forms Lemma 7 ensures $e_0 = \&x$, so $e' = x$ and $H' = H$. Subtyping Preservation Lemma 3 ensures $\Gamma_0 \vdash_{\text{rtyp}} x : \tau, r, \Gamma_1$. So it suffices to let $\Gamma_2 = \Gamma_0$.

  For DR6.6, $e' = *e'_0$ where $H; e_0 \xrightarrow{\text{r}} H'; e'_0$. So by induction, there exists a Γ_2 such that $\underline{\Gamma_2 \vdash_{\text{wf}} \Gamma_2}$, $\underline{\Gamma_2 \vdash_{\text{htyp}} H' : \Gamma_2}$, and $\Gamma_2 \vdash_{\text{rtyp}} e'_0 : \tau*, \&x, \Gamma_1$ (and if $\Gamma_0 \vdash \Gamma_1 \leq \Gamma_0$, then one such Γ_2 is Γ_0). So SR6.8A lets us derive $\underline{\Gamma_2 \vdash_{\text{rtyp}} *e'_0 : \tau, r, \Gamma_1}$ (and if $\Gamma_0 \vdash \Gamma_1 \leq \Gamma_0$, then $\Gamma_0 \vdash_{\text{rtyp}} *e'_0 : \tau, r, \Gamma_1$).
- SR6.8B: Let $e = *e_0$ and $\tau = \tau_0@$. Only rules DR6.3 and DR6.6 apply. By inversion, $\Gamma_0 \vdash_{\text{rtyp}} e_0 : \tau_0@*, \text{ALL}@, \Gamma_1$.

  For DR6.3, let $e_0 = \&x$, so $e' = x$ and $H' = H$. Subtyping Preservation Lemma 3 ensures there exists an r such that $\Gamma_1 \vdash_{\text{wf}} \tau_0@, \text{ESC}, r$ and $\Gamma_0 \vdash_{\text{rtyp}} x : \tau_0@, r, \Gamma_1$. Inverting $\Gamma_1 \vdash_{\text{wf}} \tau_0@, \text{ESC}, r$ ensures r is ALL@. So it suffices to let $\Gamma_2 = \Gamma_0$.

  For DR6.6, the argument is analogous to the argument in case SR6.8A, using rule SR6.8B in place of SR6.8A and ALL@ in place of $\&x$.
- SR6.8C: Let $e = *e_0$ and $\tau = \tau_0*$. Only rules DR6.3 and DR6.6 apply. By inversion, $\Gamma_0 \vdash_{\text{rtyp}} e_0 : \tau_0**, \text{ALL}@, \Gamma_1$.

  For DR6.3, the argument is analogous to the argument in case SR6.8B except inverting $\Gamma_1 \vdash_{\text{wf}} \tau_0*, \text{ESC}, r$ ensures r is ALL$*$.

  For DR6.6, the argument is analogous to the argument in case SR6.8A, using rule SR6.8C in place of SR6.8A and ALL@ in place of $\&x$.
- SR6.8D: This case is analogous to case SR6.8C, using int in place of τ_0*.
- SR6.9: Let $e = e_0 \| e_1$ and $\Gamma_1 = \Gamma_0 = \Gamma$. Only rules DR6.5 or DR6.6 (applied to e_0 or e_1) apply. By inversion, $\Gamma \vdash_{\text{rtyp}} e_0 : \tau', r', \Gamma$ and $\Gamma \vdash_{\text{rtyp}} e_1 : \tau, r, \Gamma$.

  For DR6.5, $e' = e_1$ and $H' = H$. So $\Gamma \vdash_{\text{rtyp}} e_1 : \tau, r, \Gamma$ and the lemma's assumptions suffice.

  For DR6.6, assume $e' = e'_0 \| e_1$ where $H; e_0 \xrightarrow{\text{r}} H'; e'_0$. (The case where $e' = e_0 \| e'_1$ is completely analogous.) By induction, $\Gamma \vdash_{\text{rtyp}} e'_0 : \tau', r', \Gamma$ and $\underline{\Gamma \vdash_{\text{htyp}} H' : \Gamma}$. So with $\Gamma \vdash_{\text{rtyp}} e_1 : \tau, r, \Gamma$, SR6.9 lets us derive $\underline{\Gamma \vdash_{\text{rtyp}} e'_0 \| e_1 : \tau, r, \Gamma}$.

- SR6.10: Let $e = (e_1{=}e_2)$. By inversion, $\Gamma_0 \vdash_{\text{ltyp}} e_1 : \tau_1, \ell, \Gamma_0$, $\Gamma_0 \vdash_{\text{rtyp}} e_2 : \tau_2, r, \Gamma_0$, $\Gamma_0 \vdash_{\text{aval}} \tau_1, \ell, r, \Gamma_1$, and $\vdash_{\text{atyp}} \tau_1, \tau_2, r$. Only rules DR6.2, DR6.6, or DR6.7 apply.

  For DR6.2, let $e = (x{=}v)$, $H = H_0, x \mapsto v'$, and $H' = H_0, x \mapsto v$. We proceed by cases on ℓ.

  If ℓ is some y, then Values Effectless Lemma 1 ensures y is x. Inverting $\Gamma_0 \vdash_{\text{aval}} \tau_1, \ell, r, \Gamma_1$ ensures $\Gamma_0 = \Gamma', x{:}\tau_1, k, r_1$ for some Γ', k, and r_1; $\Gamma_1 \vdash_{\text{wf}} \tau_1, k, r$; and $\Gamma_1 = \Gamma', x{:}\tau_1, k, r$. Inverting $\Gamma_0 \vdash_{\text{wf}} \Gamma_0$ ensures $\Gamma_0 \vdash_{\text{wf}} \Gamma'$, so Typing Well-Formedness Lemma 1 ensures $\Gamma_1 \vdash_{\text{wf}} \Gamma'$. Because $\Gamma_1 \vdash_{\text{wf}} \tau_1, k, r$, we can derive $\underline{\Gamma_1 \vdash_{\text{wf}} \Gamma_1}$. Because $\Gamma_0 \vdash_{\text{rtyp}} v : \tau_2, r, \Gamma_0$, Assignment Preservation Lemma 4 ensures $\underline{\Gamma_1 \vdash_{\text{rtyp}} v : \tau_2, r, \Gamma_1}$. Inverting $\Gamma_0 \vdash_{\text{htyp}} H : \Gamma_0$ ensures $\Gamma_0 \vdash_{\text{htyp}} H_0 : \Gamma'$. Therefore, Assignment Preservation Lemma 5 ensures $\Gamma_1 \vdash_{\text{htyp}} H_0 : \Gamma'$. So if we can show $\Gamma_1 \vdash_{\text{rtyp}} v : \tau_1, r, \Gamma_1$, then we can derive $\underline{\Gamma_1 \vdash_{\text{htyp}} H' : \Gamma_1}$. Inverting $\vdash_{\text{atyp}} \tau_1, \tau_2, r$, either $\tau_1 = \tau_2$ or $\tau_1 = \tau_0@$ and $\tau_2 = \tau_0*$ for some τ_0. If $\tau_1 = \tau_2$, we already know $\Gamma_1 \vdash_{\text{rtyp}} v : \tau_1, r, \Gamma_1$. Else $r \neq \text{NONE}$, so Canonical Forms Lemma 8 ensures the result. So letting $\Gamma_2 = \Gamma_1$ satisfies our first obligation. For our second obligation, suppose $\Gamma_0 \vdash \Gamma_1 \leq \Gamma_0$. Then Heap Subsumption Lemma 3 ensures $\Gamma_0 \vdash_{\text{htyp}} H' : \Gamma_0$. Using $\Gamma_1 \vdash_{\text{rtyp}} v : \tau_2, r, \Gamma_1$ and $\Gamma_0 \vdash_{\text{wf}} \Gamma_0$, SR6.12 lets us derive $\Gamma_0 \vdash_{\text{rtyp}} v : \tau_2, r, \Gamma_0$.

  If ℓ is ?, then inverting $\Gamma_0 \vdash_{\text{aval}} \tau_1, \ell, r, \Gamma_1$ ensures $\Gamma_1 = \Gamma_0$ and $\Gamma_0 \vdash_{\text{wf}} \tau_1, \text{ESC}, r$. We already know $\underline{\Gamma_0 \vdash_{\text{wf}} \Gamma_0}$ and $\underline{\Gamma_0 \vdash_{\text{rtyp}} v : \tau_2, r, \Gamma_0}$. It remains to show $\underline{\Gamma_0 \vdash_{\text{htyp}} H' : \Gamma_0}$. Because $\Gamma_0 \vdash_{\text{htyp}} H : \Gamma_0$, we know $\text{Dom}(\Gamma_0) = \text{Dom}(H)$. Therefore, $\Gamma_0 \vdash_{\text{ltyp}} e_1 : \tau_1, \ell, \Gamma_0$ and Values Effectless Lemma 1 ensure $x \in \text{Dom}(H)$ and $\Gamma_0(x) = \tau_1, \text{ESC}, r'$ for some r'. Because the escapedness is ESC, $\Gamma_0 \vdash_{\text{wf}} \Gamma_0$ and the rules for $\Gamma_0 \vdash_{\text{wf}} \tau_1, \text{ESC}, r'$ ensure r' is r. Therefore, $\Gamma_0 \vdash_{\text{rtyp}} v : \tau_2, r', \Gamma_0$. Inverting $\vdash_{\text{atyp}} \tau_1, \tau_2, r$, either $\tau_1 = \tau_2$ or $\tau_1 = \tau_0@$ and $\tau_2 = \tau_0*$ for some τ_0. If $\tau_1 = \tau_2$, we already know $\Gamma_0 \vdash_{\text{rtyp}} v : \tau_1, r', \Gamma_0$. For the latter, $\Gamma_0 \vdash_{\text{wf}} \tau_1, \text{ESC}, r'$ ensures $r' = \text{ALL@}$, so Canonical Forms Lemma 8 ensures $\Gamma_0 \vdash_{\text{rtyp}} v : \tau_1, r', \Gamma_0$. Finally, inverting $\Gamma_0 \vdash_{\text{htyp}} H : \Gamma_0$, ensures $\Gamma_0 \vdash_{\text{htyp}} H_0 : \Gamma'$ where $\Gamma_0 = \Gamma', x{:}\tau_1, \text{ESC}, r'$, so $\Gamma_0 \vdash_{\text{rtyp}} v : \tau_1, r', \Gamma_0$ lets us derive $\Gamma_0 \vdash_{\text{htyp}} H' : \Gamma_0$.

  For DR6.6, let $e' = (e_1{=}e_2')$ where $H; e_2 \xrightarrow{\text{r}} H'; e_2'$. So induction ensures $\Gamma_0 \vdash_{\text{rtyp}} e_2' : \tau_2, r, \Gamma_0$ and $\underline{\Gamma_0 \vdash_{\text{htyp}} H' : \Gamma_0}$. So SR6.10 ensures $\underline{\Gamma_0 \vdash_{\text{rtyp}} e_1{=}e_2' : \tau_2, r, \Gamma_1}$.

  For DR6.7, let $e' = (e_1'{=}e_2)$ where $H; e_1 \xrightarrow{\text{l}} H'; e_1'$. So induction ensures $\Gamma_0 \vdash_{\text{ltyp}} e_1' : \tau_1, \ell, \Gamma_0$ and $\underline{\Gamma_0 \vdash_{\text{htyp}} H' : \Gamma_0}$. So SR6.10 ensures $\underline{\Gamma_0 \vdash_{\text{rtyp}} e_1'{=}e_2 : \tau_2, r, \Gamma_1}$.
- SR6.11: This case follows from induction: Given $\Gamma_2 \vdash_{\text{rtyp}} e' : \tau_0@, r, \Gamma_1$,

SR6.11 lets us derive $\Gamma_2 \vdash_{\text{rtyp}} e' : \tau_0*, r, \Gamma_1$.

- SR6.12: By inversion $\Gamma_0 \vdash_{\text{rtyp}} e : \tau, r, \Gamma'$ and $\Gamma_1 \vdash \Gamma' \leq \Gamma_1$ for some Γ'. By induction, there exists a Γ_2 such that $\Gamma_2 \vdash_{\text{wf}} \Gamma_2$, $\Gamma_2 \vdash_{\text{htyp}} H' : \Gamma_2$, and $\Gamma_0 \vdash_{\text{rtyp}} e' : \tau, r, \Gamma'$, so SR6.12 lets us derive $\Gamma_2 \vdash_{\text{rtyp}} e' : \tau, r, \Gamma_1$. Furthermore, suppose $\Gamma_0 \vdash \Gamma_1 \leq \Gamma_0$. Then $\Gamma_1 \vdash \Gamma' \leq \Gamma_1$ and Abstract-Ordering Transitivity Lemma 5 ensure $\Gamma_0 \vdash \Gamma' \leq \Gamma_0$. Therefore, induction ensures we can assume Γ_2 is Γ_0.
- SR6.13: This case follows from induction: Given $\Gamma_2 \vdash_{\text{rtyp}} e' : \tau, r', \Gamma_1$ and $\Gamma_1 \vdash r' \leq r$, SR6.13 lets us derive $\Gamma_2 \vdash_{\text{rtyp}} e' : \tau, r, \Gamma_1$.
- SL6.1: This case is trivial because no rule applies.
- SL6.2A: Let $e = *e_0$. By inversion, $\Gamma_0 \vdash_{\text{rtyp}} e_0 : \tau*, \&x, \Gamma_1$. Rules DL6.1 and DL6.2 can apply.

  For DL6.1, Canonical Forms Lemma 5 ensures $e_0 = \&x$, so $e' = x$ and $H' = H$. Subtyping Preservation Lemma 1 ensures $\Gamma_0 \vdash_{\text{ltyp}} x : \tau, x, \Gamma_1$. So it suffices to let $\Gamma_2 = \Gamma_0$.

  For DL6.2, $e' = *e'_0$ where $H; e_0 \xrightarrow{\text{r}} H'; e'_0$. So by induction, there exists a Γ_2 such that $\underline{\Gamma_2 \vdash_{\text{wf}} \Gamma_2}$, $\underline{\Gamma_2 \vdash_{\text{htyp}} H' : \Gamma_2}$, and $\Gamma_2 \vdash_{\text{rtyp}} e'_0 : \tau*, \&x, \Gamma_1$ (and if $\Gamma_0 \vdash \Gamma_1 \leq \Gamma_0$, then $\underline{\text{one such } \Gamma_2 \text{ is } \Gamma_0}$). So SL6.2A lets us derive $\underline{\Gamma_2 \vdash_{\text{ltyp}} *e'_0 : \tau, \ell, \Gamma_1}$ (and if $\Gamma_0 \vdash \Gamma_1 \leq \Gamma_0$, then $\Gamma_0 \vdash_{\text{ltyp}} *e'_0 : \tau, \ell, \Gamma_1$).
- SL6.2B: Let $e = *e_0$. By inversion, $\Gamma_0 \vdash_{\text{rtyp}} e_0 : \tau*, \text{ALL@}, \Gamma_1$. Rules DL6.1 and DL6.2 can apply.

  For DL6.1, let $e_0 = \&x$, so $e' = x$ and $H' = H$. Subtyping Preservation Lemma 1 ensures $\Gamma_0 \vdash_{\text{ltyp}} x : \tau, ?, \Gamma_1$. So it suffices to let $\Gamma_2 = \Gamma_0$.

  For DL6.2, the argument is analogous to the argument in case SL6.2A, using SL6.2B in place of SL6.2A, ALL@ in place of $\&x$, and ? in place of x.
- SL6.3: This case is analogous to case SR6.12, using $\vdash_{\text{ltyp}}$ and ℓ in place of $\vdash_{\text{rtyp}}$ and r.
- SL6.4: This case follows from induction: Given $\Gamma_2 \vdash_{\text{ltyp}} e' : \tau, \ell', \Gamma_1$ and $\Gamma_1 \vdash \ell' \leq \ell$, SL6.4 lets us derive $\Gamma_2 \vdash_{\text{ltyp}} e' : \tau, \ell, \Gamma_1$.

2. The proof is by cases on the rule used to derive $V; \Gamma_0 \vdash_{\text{tst}} e : \Gamma_1; \Gamma_2$.

   - ST6.1: Inversion ensures $\Gamma_0 \vdash_{\text{rtyp}} e : \tau, 0, \Gamma_2$ and $V; \text{Dom}(\Gamma_0) \vdash_{\text{wf}} \Gamma_1$. Preservation Lemma 1 ensures all our obligations except $V; \Gamma_3 \vdash_{\text{tst}} e' : \Gamma_1; \Gamma_2$. It also ensures $\Gamma_3 \vdash_{\text{rtyp}} e' : \tau, 0, \Gamma_2$. Because $\text{Dom}(\Gamma_3) = \text{Dom}(\Gamma_0)$, we know $V; \text{Dom}(\Gamma_3) \vdash_{\text{wf}} \Gamma_1$. Therefore, ST6.1 lets us derive $V; \Gamma_3 \vdash_{\text{tst}} e' : \Gamma_1; \Gamma_2$).

- ST6.2–3: These cases are similar to case ST6.1.
- ST6.4: Inversion ensures $\Gamma_0 \vdash_{\text{ltyp}} e : \tau, x, \Gamma', x{:}\tau, \text{UNESC}, \text{ALL}*$ where $\Gamma_1 = \Gamma', x{:}\tau, \text{UNESC}, \text{ALL}@$ and $\Gamma_2 = \Gamma', x{:}\tau, \text{UNESC}, 0$. The Typing Well-Formedness Lemmas ensure $\Gamma', x{:}\tau, \text{UNESC}, \text{ALL}* \vdash_{\text{wf}} \Gamma', x{:}\tau, \text{UNESC}, \text{ALL}*$, $\Gamma_1 \vdash_{\text{wf}} \Gamma_1$, and $\Gamma_2 \vdash_{\text{wf}} \Gamma_2$. Inverting $\Gamma_0 \vdash_{\text{ltyp}} e : \tau, x, \Gamma', x{:}\tau, \text{UNESC}, \text{ALL}*$ ensures e is y for some y or $*e_0$ for some e_0. We proceed by cases on the form of e.

  If e is $*e_0$, then $H; *e_0 \xrightarrow{\text{r}} H'; e'$ with either DR6.6 or DR6.3. If the step uses DR6.6, then inspection of DL6.2 ensures $H; *e_0 \xrightarrow{\text{l}} H'; e'$. Similarly, if the step uses DR6.3, then inspection of DL6.1 ensures $H; *e_0 \xrightarrow{\text{l}} H'; e'$. Therefore, Preservation Lemma 1 ensures all our obligations except $V; \Gamma_3 \vdash_{\text{tst}} e' : \Gamma_1; \Gamma_2$. It also ensures $\Gamma_3 \vdash_{\text{ltyp}} e' : \tau, x, \Gamma', x{:}\tau, \text{UNESC}, \text{ALL}*$. So ST6.4 lets us derive $V; \Gamma_3 \vdash_{\text{tst}} e' : \Gamma_1; \Gamma_2$.

  If e is y, then Values Effectless Lemma 1 ensures e is x. Only rule DR6.1 applies, so $H' = H$ and $e' = H(x)$. Values Effectless Lemma 1 and $\Gamma_0 \vdash_{\text{ltyp}} x : \tau, x, \Gamma', x{:}\tau, \text{UNESC}, \text{ALL}*$ ensure $\Gamma', x{:}\tau, \text{UNESC}, \text{ALL}* \vdash \Gamma_0 \leq \Gamma', x{:}\tau, \text{UNESC}, \text{ALL}*$. Letting $H = H_0, x \mapsto H(x)$ and inverting $\Gamma_0 \vdash_{\text{htyp}} H : \Gamma_0$ ensures $\Gamma_0 \vdash_{\text{rtyp}} H(x) : \tau_0, r_0, \Gamma_0$ and $\Gamma_0 \vdash_{\text{htyp}} H_0 : \Gamma_0'$ where $\Gamma_0 = \Gamma_0', x{:}\tau_0, k_0, r_0$. The Abstract-Ordering Inversion Lemma ensures $\tau_0 = \tau$, $\vdash k_0 \leq \text{UNESC}$ (i.e., $k_0 = \text{UNESC}$), and $\Gamma', x{:}\tau, \text{UNESC}, \text{ALL}* \vdash r_0 \leq \text{ALL}*$. So the Abstract Antisymmetry Lemmas ensure r_0 is not NONE. So Canonical Forms Lemma 9 ensures $H(x)$ is not junk. Furthermore, Typing Well-Formedness Lemma 2 ensures $\text{Dom}(\Gamma') = \text{Dom}(\Gamma_0')$. So we have established the hypotheses necessary for Heap Subsumption Lemma 3 to show $\Gamma', x{:}\tau, \text{UNESC}, \text{ALL}* \vdash_{\text{htyp}} H_0 : \Gamma'$. Therefore, Assignment Preservation Lemma 5 ensures $\Gamma_1 \vdash_{\text{htyp}} H_0 : \Gamma'$ and $\Gamma_2 \vdash_{\text{htyp}} H_0 : \Gamma'$. We proceed by cases on $H(x)$.

  If $H(x)$ is 0, let $\Gamma_3 = \Gamma_2$. Depending on τ, SR6.2 or SR6.3 lets us derive $\Gamma_2 \vdash_{\text{rtyp}} 0 : \tau, 0, \Gamma_2$. (We know τ cannot have the form $\tau'@$ because inverting $\Gamma_2 \vdash_{\text{wf}} \Gamma_2$ ensures $\Gamma_2 \vdash_{\text{wf}} \tau, \text{UNESC}, \text{ALL}*$.) So $H(x) = 0$ and $\Gamma_2 \vdash_{\text{htyp}} H_0 : \Gamma'$ means we can derive $\underline{\Gamma_2 \vdash_{\text{htyp}} H : \Gamma_2}$. Because $\text{Dom}(\Gamma_1) = \text{Dom}(\Gamma_2)$, we can derive $V; \text{Dom}(\Gamma_2) \vdash_{\text{wf}} \Gamma_1$. So ST6.1 lets us derive $\underline{V; \Gamma_2 \vdash_{\text{tst}} H(x) : \Gamma_1; \Gamma_2}$. Because $\underline{\Gamma_2 \vdash_{\text{wf}} \Gamma_2}$, the lemma holds in this case.

  If $H(x)$ is some $i \neq 0$, let $\Gamma_3 = \Gamma_1$. A trivial induction on $\Gamma_0 \vdash_{\text{rtyp}} H(x) : \tau, r_0, \Gamma_0$ ensures τ is int. So SR6.4 lets us derive $\Gamma_1 \vdash_{\text{rtyp}} i : \tau, \text{ALL}@, \Gamma_1$. So $H(x) = i$ and $\Gamma_1 \vdash_{\text{htyp}} H_0 : \Gamma'$ means we can derive $\Gamma_1 \vdash_{\text{htyp}} H : \Gamma_1$. Because $\text{Dom}(\Gamma_2) = \text{Dom}(\Gamma_1)$, we can derive $V; \text{Dom}(\Gamma_1) \vdash_{\text{wf}} \Gamma_2$. So ST6.3 lets us derive $\underline{V; \Gamma_1 \vdash_{\text{tst}} H(x) : \Gamma_1; \Gamma_2}$. Because $\underline{\Gamma_1 \vdash_{\text{wf}} \Gamma_1}$, the lemma holds in this case.

If $H(x)$ is some $\&y$, let $\Gamma_3 = \Gamma_1$. If we can show $\Gamma_1 \vdash_{\text{rtyp}} \&y : \tau, \text{ALL@}, \Gamma_1$, the argument continues as when $H(x)$ is some $i \neq 0$. A trivial induction on $\Gamma_0 \vdash_{\text{rtyp}} H(x) : \tau, r_0, \Gamma_0$ ensures τ is not int. Values Effectless Lemma 2 ensures there exist τ', k', and r' such that $\Gamma_0(y) = \tau', k', r'$, τ is $\tau'*$ or $\tau'@$, and $\Gamma_0 \vdash \&y \leq r_0$. So Typing Well-Formedness Lemma 2 and the Abstract-Ordering Inversion Lemma ensure Γ_1 also maps y to τ'. So SL6.1 and SR6.7A let us derive $\Gamma_1 \vdash_{\text{rtyp}} \&y : \tau'@, \&y, \Gamma_1$. So by possibly using SR6.11, we know $\Gamma_1 \vdash_{\text{rtyp}} \&y : \tau, \&y, \Gamma_1$. So SR6.13 lets us conclude $\Gamma_1 \vdash_{\text{rtyp}} \&y : \tau, \text{ALL@}, \Gamma_1$ if $\Gamma_1(y) = \tau_1, \text{ESC}, r_1$ for some τ_1 and r_1.

Because $\Gamma', x{:}\tau, \text{UNESC}, \text{ALL}* \vdash \Gamma_0 \leq \Gamma', x{:}\tau, \text{UNESC}, \text{ALL}*$ and $\Gamma_0 \vdash \&y \leq r_0$, Abstract-Ordering Transitivity Lemma 3 ensures $\Gamma', x{:}\tau, \text{UNESC}, \text{ALL}* \vdash \&y \leq r_0$. Therefore, because $\Gamma', x{:}\tau, \text{UNESC}, \text{ALL}* \vdash r_0 \leq \text{ALL}*$, we can derive $\Gamma', x{:}\tau, \text{UNESC}, \text{ALL}* \vdash \&y \leq \text{ALL}*$. Therefore, Assignment Preservation Lemma 2 ensures $\Gamma_1 \vdash \&y \leq \text{ALL}*$. (We showed the well-formedness hypotheses of this lemma above.) Therefore, Abstract-Ordering Antisymmetry Lemma 5 ensures $\Gamma_1(y) = \tau_1, \text{ESC}, r_1$, as required.

- ST6.5: Inversion ensures $\Gamma_0 \vdash_{\text{rtyp}} e : \tau, \text{ALL}*, \Gamma_1$ and $\Gamma_1 = \Gamma_2$. Preservation Lemma 1 ensures all our obligations except $V; \Gamma_3 \vdash_{\text{tst}} e' : \Gamma_1; \Gamma_1$. It also ensures $\Gamma_3 \vdash_{\text{rtyp}} e' : \tau, \text{ALL}*, \Gamma_1$ (therefore, ST6.5 lets us derive $V; \Gamma_3 \vdash_{\text{tst}} e' : \Gamma_1; \Gamma_1$).

3. The proof is by induction on the statement-typing derivation that inversion of $\vdash_{\text{prog}} V_0; H; s : \Gamma_1$ ensures (i.e., $V_0'; \Gamma_0 \vdash_{\text{styp}} s : \Gamma_1$ where $V_0'' \subseteq V_0'$), proceeding by cases on the last rule used.

   - SS6.1: Let $s = e$. Inversion ensures $\Gamma_0 \vdash_{\text{rtyp}} e : \tau, r, \Gamma_1$ for some τ and r. Only DS6.7 applies, so $H; e \xrightarrow{\text{r}} H'; e'$. So Preservation Lemma 1 ensures there exists a Γ_2 such that $\text{Dom}(\Gamma_2) = \text{Dom}(\Gamma_0)$, $\underline{\Gamma_2 \vdash_{\text{wf}} \Gamma_2}$, $\underline{\Gamma_2 \vdash_{\text{htyp}} H' : \Gamma_2}$, and $\Gamma_0 \vdash_{\text{rtyp}} e' : \tau, r, \Gamma_1$. So SS6.1 ensures $\underline{V_0'; \Gamma_0 \vdash_{\text{styp}} e' : \Gamma_1}$. Inverting $\vdash_{\text{v}} e : V_0''$ ensures $V_0'' = \cdot$, so we can derive $\underline{\vdash_{\text{v}} e' : V_0''}$. By assumption, $\underline{V_0'' \subseteq V_0'}$. Because $\text{Dom}(\Gamma_2) = \text{Dom}(\Gamma_0)$, we $\underline{\text{know } \text{Dom}(H') = \text{Dom}(H)}$. $\underline{\text{So } V_0' \cap \text{Dom}(H') = \emptyset}$. So the assumption $V_0 \supseteq V_0' \cup \text{Dom}(H)$ ensures $\underline{V_0 \supseteq V_0' \cup \text{Dom}(H')}$. Letting $V_1 = V_0$, the underlined hypotheses ensure $\underline{\vdash_{\text{prog}} V_1; H'; s' : \Gamma_1}$. The other results are trivial because $V_1 = V_0$, $H' = H$, and $V_1'' = V_0''$.
   - SS6.2: This case is trivial because no rule applies.

- SS6.3: Let $s = s_1; s_2$. Inversion ensures there exists a Γ' such that $V_0'; \Gamma_0 \vdash_{\text{styp}} s_1 : \Gamma'$, $V_0' - V_{0\text{A}}''; \Gamma' \vdash_{\text{styp}} s_2 : \Gamma_1$, $\vdash_{\text{v}} s_1 : V_{0\text{A}}''$, $\vdash_{\text{v}} s_2 : V_{0\text{B}}''$, $V_{0\text{A}} \cap V_{0\text{B}} = \emptyset$, $V_0'' = V_{0\text{A}}'' \cup V_{0\text{B}}''$, and $V_0'' \subseteq V_0'$. Only DS6.2, DS6.3, and DS6.8 can apply.

  For DS6.2, $s_1 = v$ for some v and $V_0; H; s \xrightarrow{\text{s}} V_0; H; s_2$. So inverting $\vdash_{\text{v}} v; s_2 : V_0''$ ensures $\vdash_{\text{v}} s_2 : V_0''$. So letting $V_1 = V_0$ and $V_1'' = V_0''$, the result follows from the Value Elimination Lemma.

  For DS6.3, $s_1 = \mathsf{return}$ and $V_0; H; s \xrightarrow{\text{s}} V_0; H; \mathsf{return}$. Trivially, we know $\underline{\vdash_{\text{v}} \mathsf{return} : \emptyset}$, $\underline{\emptyset \subseteq V_0'}$, and $\emptyset \cap V_0 \subseteq V_0''$. So given the assumptions and the underlined results, we just need to show $V_0'; \Gamma_0 \vdash_{\text{styp}} \mathsf{return} : \Gamma_1$. Applying Typing Well-Formedness Lemma 5 to $V_0'; \Gamma_0 \vdash_{\text{styp}} s_1 : \Gamma'$ and $V_0' - V_{0\text{A}}''; \Gamma' \vdash_{\text{styp}} s_2 : \Gamma_1$ ensures $\text{Dom}(\Gamma') \subseteq \text{Dom}(\Gamma_0) \cup V_0'$, $\text{Dom}(\Gamma_1) \subseteq \text{Dom}(\Gamma') \cup (V_0' - V_{0\text{A}}'')$, and $\Gamma_1 \vdash_{\text{wf}} \Gamma_1$. Therefore, $\text{Dom}(\Gamma_1) \subseteq \text{Dom}(\Gamma_0) \cup V_0'$. If $\text{Dom}(\Gamma_0) \subseteq \text{Dom}(\Gamma_1)$, then $V_0'; \text{Dom}(\Gamma_0) \vdash_{\text{wf}} \Gamma_1$, so SS6.2 lets us derive $V_0'; \Gamma_0 \vdash_{\text{styp}} \mathsf{return} : \Gamma_1$. Else $\text{Dom}(\Gamma_1) \subseteq \text{Dom}(\Gamma_0)$. In this case, let $\Gamma_0 = \Gamma_{0\text{A}}\Gamma_{0\text{B}}$ where $\text{Dom}(\Gamma_1) = \text{Dom}(\Gamma_{0\text{A}})$. Inverting $\Gamma_0 \vdash_{\text{wf}} \Gamma_0$ ensures $\Gamma_0 \vdash_{\text{wf}} \Gamma_{0\text{B}}$. Therefore, $\Gamma_1 \vdash_{\text{wf}} \Gamma_1$, Typing Well-Formedness Lemma 1, and Weakening Lemma 3 ensure $\Gamma_1\Gamma_{0\text{B}} \vdash_{\text{wf}} \Gamma_1\Gamma_{0\text{B}}$. Therefore, $V_0'; \text{Dom}(\Gamma_0) \vdash_{\text{wf}} \Gamma_1\Gamma_{0\text{B}}$, so SS6.2 lets us derive $V_0'; \Gamma_0 \vdash_{\text{styp}} \mathsf{return} : \Gamma_1\Gamma_{0\text{B}}$. Therefore, $\Gamma_1 \vdash_{\text{wf}} \Gamma_1$ and SS6.8 let us derive $V_0'; \Gamma_0 \vdash_{\text{styp}} \mathsf{return} : \Gamma_1$.

  For DS6.8, $V_0; H; s_1 \xrightarrow{\text{s}} V_1; H'; s_1'$ and $s' = s_1'; s_2$. Because $V_{0\text{A}}'' \subseteq V_0'$, $V_0'; \Gamma_0 \vdash_{\text{styp}} s_1 : \Gamma'$ and induction ensure $\vdash_{\text{prog}} V_1; H'; s_1' : \Gamma'$, $V_1 \supseteq V_0$, $\text{Dom}(H') \subseteq \text{Dom}(H) \cup V_{0\text{A}}''$, and $V_{1\text{A}}'' \cap V_0 \subseteq V_{0\text{A}}''$ where $\vdash_{\text{v}} s_1' : V_{1\text{A}}''$. Inverting $\vdash_{\text{prog}} V_1; H'; s_1' : \Gamma'$ means $\underline{\Gamma_2 \vdash_{\text{htyp}} H' : \Gamma_2}$, $V_{1\text{A}}'; \Gamma_2 \vdash_{\text{styp}} s_1' : \Gamma'$, $\underline{\Gamma_2 \vdash_{\text{wf}} \Gamma_2}$, $V_{1\text{A}}'' \subseteq V_{1\text{A}}'$, $V_{1\text{A}}' \cap \text{Dom}(H') = \emptyset$, and $\underline{V_1 \supseteq V_{1\text{A}}' \cup \text{Dom}(H')}$.
  Let $V_1'' = V_{1\text{A}}'' \cup V_{0\text{B}}''$ and $V_1' = V_{1\text{A}}' \cup (V_0' - V_{0\text{A}}'')$.
  Weakening Lemma 11 and $V_{1\text{A}}'; \Gamma_2 \vdash_{\text{styp}} s_1' : \Gamma'$ ensure $V_1'; \Gamma_2 \vdash_{\text{styp}} s_1' : \Gamma'$. Because $V_{1\text{A}}'' \cap V_0 \subseteq V_{0\text{A}}''$ and $V_0' \subseteq V_0$, we can rewrite $V_0' - V_{0\text{A}}''; \Gamma' \vdash_{\text{styp}} s_2 : \Gamma_1$ as $(V_0' - V_{0\text{A}}'') - V_{1\text{A}}''; \Gamma' \vdash_{\text{styp}} s_2 : \Gamma_1$. So Weakening Lemma 11 ensures $V_1' - V_{1\text{A}}''; \Gamma' \vdash_{\text{styp}} s_2 : \Gamma_1$. So SS6.3 lets us derive $\underline{V_1'; \Gamma_2 \vdash_{\text{styp}} s_1'; s_2 : \Gamma_1}$.
  Because $V_{0\text{B}}'' \subseteq V_0$, $V_{0\text{A}}'' \cap V_{0\text{B}}'' = \emptyset$, and $V_{1\text{A}}'' \cap V_0 \subseteq V_{0\text{A}}''$, we know $V_{1\text{A}}'' \cap V_{0\text{B}}'' = \emptyset$. So $\vdash_{\text{v}} s_1' : V_{1\text{A}}''$ and $\vdash_{\text{v}} s_2 : V_{0\text{B}}''$ lets us derive $\underline{\vdash_{\text{v}} s_1; s_2 : V_1''}$
  Because $V_{1\text{A}}'' \subseteq V_{1\text{A}}'$ and $V_{0\text{B}}'' \subseteq V_0'$, and $V_{0\text{A}}'' \cap V_{0\text{B}}'' = \emptyset$, we know $\underline{V_1'' \subseteq V_1'}$.
  Because $\text{Dom}(H') \subseteq \text{Dom}(H) \cup V_{0\text{A}}''$, $V_{1\text{A}}' \cap \text{Dom}(H') = \emptyset$, and $V_0' \cap \text{Dom}(H) = \emptyset$, we know $\underline{V_1' \cap \text{Dom}(H') = \emptyset}$. Because $V_1 \supseteq V_0 \supseteq V_0'$ and $V_1 \supseteq V_{1\text{A}}' \cup \text{Dom}(H')$, we know $\underline{V_1 \supseteq V_1' \cup \text{Dom}(H')}$. The underlined results ensure $\vdash_{\text{prog}} V_1; H'; s' : \Gamma_1$.
  As for the other obligations, induction showed $V_1 \supseteq V_0$. It also showed $\text{Dom}(H') \subseteq \text{Dom}(H) \cup V_{0\text{A}}''$, so $V_{0\text{A}}'' \subseteq V_0''$ suffices to show $\text{Dom}(H') \subseteq$

$\mathrm{Dom}(H) \cup V_0''$. Similarly, $V_{1\mathrm{A}}'' \cap V_0 \subseteq V_{0\mathrm{A}}''$ ensures $(V_{1\mathrm{A}}'' \cup V_{0\mathrm{B}}'') \cap V_0 \subseteq V_{0\mathrm{A}}'' \cup V_{0\mathrm{B}}''$ (because $V_{0\mathrm{B}}'' \subseteq V_0$).

- SS6.4: Let $s = \mathsf{while}\ e\ s_1$. Inversion ensures $V_0; \Gamma_0 \vdash_{\mathrm{tst}} e : \Gamma'; \Gamma_1$ and $V_0; \Gamma' \vdash_{\mathrm{styp}} s_1 : \Gamma_0$ for some Γ'. Furthermore, $\vdash_{\mathrm{v}} s_1 : V_0''$. Only rule DS6.6 applies, so $s' = \mathsf{if}\ e\ (s_1'; \mathsf{while}\ e\ s_1)\ 0$, $H' = H$, and $V_1 = V_0 \cup V_A$ where $s_1' = \mathrm{rename}(M, s_1)$, $\mathrm{Dom}(M) = \mathrm{Dom}(V_0'')$, $V_0 \vdash_{\mathrm{wf}} M$, and $\vdash_{\mathrm{v}} s_1' : V_A$.

  Because $V_0'' \subseteq V_0'$ and $V_0 \supseteq V_0' \cup \mathrm{Dom}(H)$, we know $V_0'' \subseteq V_0' \subseteq V_0$. So the Systematic Renaming Lemma ensures $V_0 \cup V_A; \mathrm{rename}(M, \Gamma') \vdash_{\mathrm{styp}} s_1' : \mathrm{rename}(M, \Gamma_0)$. The assumption $\Gamma_0 \vdash_{\mathrm{htyp}} H : \Gamma_0$ ensures $\mathrm{Dom}(\Gamma_0) = \mathrm{Dom}(H)$. So $V_0' \cap \mathrm{Dom}(H) = \emptyset$, $V_0'' \subseteq V_0'$, and $\mathrm{Dom}(M) = \mathrm{Dom}(V_0'')$ ensure $\mathrm{Dom}(M) \cap \mathrm{Dom}(\Gamma_0) = \emptyset$. So the Useless Renaming Lemma ensures $V_0 \cup V_A; \mathrm{rename}(M, \Gamma') \vdash_{\mathrm{styp}} s_1' : \Gamma_0$, i.e., $V_1; \mathrm{rename}(M, \Gamma') \vdash_{\mathrm{styp}} s_1' : \Gamma_0$.

  Rules SR6.2 and SS6.1 let us derive $V_1; \Gamma_1 \vdash_{\mathrm{styp}} 0 : \Gamma_1$. Because $V_0; \Gamma_0 \vdash_{\mathrm{styp}} \mathsf{while}\ e\ s_1 : \Gamma_1$, we can write the equivalent $V_1 - V_A; \Gamma_0 \vdash_{\mathrm{styp}} \mathsf{while}\ e\ s_1 : \Gamma_1$. Therefore, if we assume $V_1; \Gamma_0 \vdash_{\mathrm{tst}} e : \mathrm{rename}(M, \Gamma'); \Gamma_1$, then we have the following derivation:

$$\frac{\begin{array}{l} V_1; \mathrm{rename}(M, \Gamma') \vdash_{\mathrm{styp}} s_1' : \Gamma_0 \\ V_1 - V_A; \Gamma_0 \vdash_{\mathrm{styp}} \mathsf{while}\ e\ s_1 : \Gamma_1 \\ \vdash_{\mathrm{v}} s_1' : V_A \end{array}}{V_1; \mathrm{rename}(M, \Gamma') \vdash_{\mathrm{styp}} s_1'; \mathsf{while}\ e\ s_1 : \Gamma_1}$$

$$\frac{\begin{array}{l} V_1; \Gamma_0 \vdash_{\mathrm{tst}} e : \mathrm{rename}(M, \Gamma'); \Gamma_1 \\ V_1; \mathrm{rename}(M, \Gamma') \vdash_{\mathrm{styp}} s_1'; \mathsf{while}\ e\ s_1 : \Gamma_1 \\ V_1; \Gamma_1 \vdash_{\mathrm{styp}} 0 : \Gamma_1 \end{array}}{V_1; \Gamma_0 \vdash_{\mathrm{styp}} \mathsf{if}\ e\ (s_1'; \mathsf{while}\ e\ s_1)\ 0 : \Gamma_1}$$

  So we need $V_1; \Gamma_0 \vdash_{\mathrm{tst}} e : \mathrm{rename}(M, \Gamma'); \Gamma_1$ to conclude $V_1; \Gamma_0 \vdash_{\mathrm{styp}} s' : \Gamma_1$. We proceed by cases on the derivation of $V_1; \Gamma_0 \vdash_{\mathrm{tst}} e : \Gamma'; \Gamma_1$ (which exists because of Weakening Lemma 10 and $V_0; \Gamma_0 \vdash_{\mathrm{tst}} e : \Gamma'; \Gamma_1$). For cases ST6.2–ST6.5, inversion and Typing Well-Formedness Lemma 3 ensure $\Gamma' \vdash_{\mathrm{wf}} \Gamma'$ and $\mathrm{Dom}(\Gamma_0) = \mathrm{Dom}(\Gamma')$. Therefore, $\mathrm{Dom}(M) \cap \mathrm{Dom}(\Gamma_0) = \emptyset$ ensures $\mathrm{Dom}(M) \cap \mathrm{Dom}(\Gamma_1) = \emptyset$, so the Useless Renaming Lemma ensures $\mathrm{rename}(M, \Gamma') = \Gamma'$. So $V_1; \Gamma_0 \vdash_{\mathrm{tst}} e : \Gamma'; \Gamma_1$ suffices. For case ST6.1, inversion ensures it suffices to show $V_1; \mathrm{Dom}(\Gamma_0) \vdash_{\mathrm{wf}} \mathrm{rename}(M, \Gamma')$. An omitted Systematic Renaming Lemma ensures $\mathrm{rename}(M, \Gamma') \vdash_{\mathrm{wf}} \mathrm{rename}(M, \Gamma')$ because $\Gamma' \vdash_{\mathrm{wf}} \Gamma'$ and $V_0 \vdash_{\mathrm{wf}} M$. By inversion,

$\mathrm{Dom}(\Gamma_0) \subseteq \mathrm{Dom}(\Gamma') \subseteq V_0 \cup \mathrm{Dom}(\Gamma_0)$, so $V_1 = V_0 \cup V_A$ ensures $\mathrm{Dom}(\Gamma_0) \subseteq \mathrm{Dom}(\Gamma') \subseteq V_1 \cup \mathrm{Dom}(\Gamma_0)$. So we can derive $V_1; \mathrm{Dom}(\Gamma_0) \vdash_{\mathrm{wf}} \mathrm{rename}(M, \Gamma')$.

Because $\vdash_{\mathrm{v}} s'_1 : V_A$ and $\vdash_{\mathrm{v}} s : V''_0$, we can derive $\underline{\vdash_{\mathrm{v}} s' : V_A \cup V''_0}$. Because $V_0 \supseteq V'_0 \cup \mathrm{Dom}(H)$ (and therefore $V_0 \supseteq V''_0$), the Systematic Renaming Lemma ensures $V_A \cap V_0 = \emptyset$. Therefore, because $V''_0 \cap \mathrm{Dom}(H) = \emptyset$, we know $\underline{V_A \cup V''_0 \cap \mathrm{Dom}(H) = \emptyset}$ and $\underline{V_0 \cup V_A \supseteq V''_0 \cup V_A \cup \mathrm{Dom}(H)}$. By assumption, $\underline{\Gamma_0 \vdash_{\mathrm{wf}} \Gamma_0}$. The underlined results let us derive $\vdash_{\mathrm{prog}} V_0 \cup V_A; H; s' : \Gamma_1$. Furthermore, $V_A \cap V_0 = \emptyset$ ensures $(V''_0 \cup V_A) \cap V_0 \subseteq V''_0$.

- SS6.5: Let $s = \mathsf{if}\ e\ s_1\ s_2$. Inversion ensures $V'_0; \Gamma_0 \vdash_{\mathrm{tst}} e : \Gamma_A; \Gamma_B$, $V'_0; \Gamma_A \vdash_{\mathrm{styp}} s_1 : \Gamma_1$, and $V'_0; \Gamma_B \vdash_{\mathrm{styp}} s_2 : \Gamma_1$ for some Γ_A and Γ_B. Furthermore, $\vdash_{\mathrm{v}} s_1 : V_A$, $\vdash_{\mathrm{v}} s_2 : V_B$, and $V''_0 = V_A \cup V_B$. Only rules DS6.4, DS6.5, and DS6.7 can apply.

  For DS6.4, let $e = 0$, $s' = s_2$, and $H' = H$. Because $V_B \subseteq V''_0$ and $V''_0 \subseteq V'_0$, we know $V_B \subseteq V'_0$. Therefore, the Value Elimination Lemma ensures $\vdash_{\mathrm{prog}} V_0; H; s_2 : \Gamma_1$ if $V'_0; \Gamma_0 \vdash_{\mathrm{styp}} 0 : \Gamma_B$. We show this result by cases on the derivation of $V'_0; \Gamma_0 \vdash_{\mathrm{tst}} 0 : \Gamma_A; \Gamma_B$. Case ST6.1 follows from inversion and SS6.1. Cases ST6.2 and ST6.3 cannot apply because the Canonical Forms Lemma ensure there is no Γ', τ, and x such that $\Gamma_0 \vdash_{\mathrm{rtyp}} 0 : \tau, \&x, \Gamma'$ or $\Gamma_0 \vdash_{\mathrm{rtyp}} 0 : \tau, \mathrm{ALL@}, \Gamma'$. Case ST6.4 cannot apply because a trivial induction shows 0 is not a left-expression. Case ST6.5 follows from inversion and SS6.1. The other obligations are trivial because $V_1 = V_0$, $H' = H$, and $V''_1 = V_B \subseteq V''_0$.

  Case DS6.5 is analogous to case DS6.4, where e is some v that is neither 0 nor junk. We use s_1 in place of s_2, V_A in place of V_B, and Γ_A in place of Γ_B. We use the Canonical Forms Lemmas to ensure case ST6.1 does not apply. Cases ST6.2 and ST6.3 follow from inversion and SS6.1.

  For DS6.7, $s' = \mathsf{if}\ e'\ s_1\ s_2$ and $H; e \xrightarrow{\mathrm{r}} H'; e'$. Preservation Lemma 2 ensures there exists a Γ_2 such that $\mathrm{Dom}(\Gamma_2) = \mathrm{Dom}(\Gamma_0)$, $\underline{\Gamma_2 \vdash_{\mathrm{wf}} \Gamma_2}$, $\underline{\Gamma_2 \vdash_{\mathrm{htyp}} H' : \Gamma_2}$, and $V'_0; \Gamma_0 \vdash_{\mathrm{tst}} e' : \Gamma_A; \Gamma_B$. So SS6.5 lets us derive $\underline{V'_0; \Gamma_2 \vdash_{\mathrm{styp}} \mathsf{if}\ e'\ s_1\ s_2 : \Gamma_1}$. Because $\vdash_{\mathrm{v}} s_1 : V_A$ and $\vdash_{\mathrm{v}} s_2 : V_B$, we can derive $\underline{\vdash_{\mathrm{v}} \mathsf{if}\ e'\ s_1\ s_2 : V''_0}$. By assumption, $\underline{V''_0 \subseteq V'_0}$. Because $\mathrm{Dom}(\Gamma_2) = \mathrm{Dom}(\Gamma_0)$, we know $\mathrm{Dom}(H') = \mathrm{Dom}(H)$. So $\underline{V'_0 \cap \mathrm{Dom}(H') = \emptyset}$. So the assumption $V_0 \supseteq V'_0 \cup \mathrm{Dom}(H)$ ensures $\underline{V_0 \supseteq V'_0 \cup \mathrm{Dom}(H')}$. Letting $V_1 = V_0$, the underlined hypotheses ensure $\underline{\vdash_{\mathrm{prog}} V_1; H'; s' : \Gamma_1}$. The other obligations are trivial because $V_1 = V_0$, $\mathrm{Dom}(H') = \mathrm{Dom}(H)$, and $V''_1 = V''_0$.

- SS6.6: Let $s = \tau\ x$ and $\Gamma_1 = \Gamma_0, x{:}\tau, \mathrm{UNESC}, \mathrm{NONE}$. Only DS6.1 applies,

so $s' = 0$ and $H' = H, x \mapsto \mathsf{junk}$. By assumption, $\Gamma_0 \vdash_{\text{htyp}} H : \Gamma_0$. Trivially, $\Gamma_1 \vdash_{\text{wf}} \tau, \text{UNESC}, \text{NONE}$. So by a trivial inductive argument over the derivation of $\Gamma_0 \vdash_{\text{htyp}} H : \Gamma_0$, using Weakening Lemma 7, we know $\Gamma_1 \vdash_{\text{htyp}} H : \Gamma_0$. Rule SR6.1 lets us derive $\Gamma_1 \vdash_{\text{rtyp}} \mathsf{junk} : \tau, \text{NONE}, \Gamma_1$. So we can derive $\Gamma_1 \vdash_{\text{htyp}} H' : \Gamma_1$. Rules SR6.2 and SS6.1 let us derive $\emptyset; \Gamma_1 \vdash_{\text{styp}} 0 : \Gamma_1$. Typing Well-Formedness Lemma 5 and the assumptions ensure $\Gamma_1 \vdash_{\text{wf}} \Gamma_1$. Trivially, $\vdash_{\text{V}} 0 : \emptyset$, $\emptyset \subseteq \emptyset$, and $\emptyset \cap \text{Dom}(H') = \emptyset$. Inverting $\vdash_{\text{V}} \tau\ x : V_0''$ ensures $V_0'' = \cdot, x$. So $V_0'' \subseteq V_0'$ and $V_0 \supseteq V_0' \cup \text{Dom}(H)$ ensure $V_0 \supseteq \emptyset \cup \text{Dom}(H')$. The underlined results ensure $\vdash_{\text{prog}} V_0; H'; s' : \Gamma_1$. Trivially, $V_0 \supseteq V_0$, $\text{Dom}(H') \subseteq \text{Dom}(H) \cup V_0''$, and $\emptyset \cap V_0 \subseteq V_0''$.

- SS6.7: Inverting $V_0; \Gamma_0 \vdash_{\text{styp}} s : \Gamma_1$ ensures $V_0; \Gamma_0 \vdash_{\text{styp}} s : \Gamma'$, $\Gamma_1 \vdash \Gamma' \leq \Gamma_1$, and $\Gamma_1 \vdash_{\text{wf}} \Gamma_1$ for some Γ'. So $\vdash_{\text{prog}} V_0; H; s : \Gamma'$. So induction ensures $\vdash_{\text{prog}} V_1; H'; s' : \Gamma'$, $V_1 \supseteq V_0$, $\text{Dom}(H') \subseteq \text{Dom}(H) \cup V_0''$, and $V_1'' \cap V_0 \subseteq V_0''$ where $\vdash_{\text{V}} s' : V_1''$. So SS6.7 lets us derive $\vdash_{\text{prog}} V_1; H'; s' : \Gamma_1$.
- SS6.8: Inverting $V_0; \Gamma_0 \vdash_{\text{styp}} s : \Gamma_1$ ensures $V_0; \Gamma_0 \vdash_{\text{styp}} s : \Gamma_1\Gamma'$ and $\Gamma_1 \vdash_{\text{wf}} \Gamma_1$ for some Γ'. So $\vdash_{\text{prog}} V_0; H; s : \Gamma'$. So induction ensures $\vdash_{\text{prog}} V_1; H'; s' : \Gamma_1\Gamma'$, $V_1 \supseteq V_0$, $\text{Dom}(H') \subseteq \text{Dom}(H) \cup V_0''$, and $V_1'' \cap V_0 \subseteq V_0''$ where $\vdash_{\text{V}} s' : V_1''$. So SS6.8 lets us derive $\vdash_{\text{prog}} V_1; H'; s' : \Gamma_1$.

Lemma D.15 (Progress). *Suppose* $\Gamma_0 \vdash_{\text{htyp}} H : \Gamma_0$ *and* $\Gamma_0 \vdash_{\text{wf}} \Gamma_0$.

1. *If* $\Gamma_0 \vdash_{\text{ltyp}} e : \tau, \ell, \Gamma_1$, *then* e *is* x *for some* x *or there exist* H' *and* e' *such that* $H; e \xrightarrow{\text{l}} H'; e'$.
   If $\Gamma_0 \vdash_{\text{rtyp}} e : \tau, r, \Gamma_1$, *then* e *is* v *for some* v *or there exist* H' *and* e' *such that* $H; e \xrightarrow{\text{r}} H'; e'$.

2. *If* $V; \Gamma_0 \vdash_{\text{tst}} e : \Gamma_1; \Gamma_2$, *then* e *is* v *for some* v *that is not* junk *or there exist* H' *and* e' *such that* $H; e \xrightarrow{\text{r}} H'; e'$.

3. *If* $V; \Gamma_0 \vdash_{\text{styp}} s : \Gamma_1$, *then* s *is* v *for some* v, *or* s *is* return, *or there exist* V', H', *and* s' *such that* $V; H; s \xrightarrow{\text{s}} V'; H'; s'$.

Proof:

1. The proof is by simultaneous induction on the assumed typing derivations, proceeding by cases on the last rule used:

   - SR6.1–4: These cases are trivial because e is a value.
   - SR6.5: Rule DR6.4 applies.
   - SR6.6: Because $\Gamma_0 \vdash_{\text{htyp}} H : \Gamma_0$ and $x \in \Gamma_0$, we know $x \in \text{Dom}(H)$. So rule DR6.1 applies.

- SR6.7A–B: Let $e = \&e_0$. By induction, e_0 is some x (in which case e is a value), or there exists an e_0' such that $H; e_0 \xrightarrow{\text{l}} H'; e_0'$ (in which case DR6.7 applies).
- SR6.8A–D: Let $e = *e_0$. By induction, either e_0 is some value or there exists an e_0' such that $H; e_0 \xrightarrow{\text{r}} H'; e_0'$. In the latter case, rule DR6.6 applies. In the former case, Canonical Forms Lemmas 5 and 6 ensure e_0 is $\&x$ for some x, so rule DR6.3 applies.
- SR6.9: Let $e = e_1 \| e_2$. By induction, if e_1 or e_2 is not a value, then DR6.6 applies. If e_1 and e_2 are values, then DR6.5 applies.
- SR6.10: Let $e = (e_1{=}e_2)$. By induction, if e_1 is not some x, then DR6.7 applies. By induction, if e_2 is not a value, then DR6.6 applies. If $e_1 = x$ and $e_2 = v$, then DR6.2 applies if $x \in \text{Dom}(H)$. Values Effectless Lemma 1 ensures $x \in \text{Dom}(\Gamma_0)$, so $\Gamma_0 \vdash_{\text{htyp}} H : \Gamma_0$ ensures $x \in \text{Dom}(H)$.
- SR6.11–13: These cases follow from induction.
- SL6.1: This case is trivial because e is some x.
- SL6.2A–B: Let $e = *e_0$. By induction, either e_0 is some value or there exists an e_0' such that $H; e_0 \xrightarrow{\text{r}} H'; e_0'$. In the latter case, rule DL6.2 applies. In the former case, Canonical Forms Lemmas 5 and 6 ensure e_0 is $\&x$ for some x, so rule DL6.1 applies.
- SL6.3–4: These cases follow from induction.

2. The proof is by cases on the assumed typing derivation. In each case, inversion ensures e is the subject of a right-expression typing derivation. (In case ST6.4, we need Subtyping Preservation Lemma 2 for this fact.) So the previous lemma ensures e can take a step or is some value. In the latter case, the abstract rvalues in the $\vdash_{\text{rtyp}}$ hypotheses and Canonical Forms Lemma 9 ensure the value is not junk.

3. The proof is by induction on the assumed typing derivation, proceeding by cases on the last rule used:

   - SS6.1: This case follows from rule DS6.7 and Progress Lemma 1.
   - SS6.2: This case is trivial because s is return.
   - SS6.3: Let $s = s_1; s_2$. Because s_1 is well-typed, induction ensures s_1 is v or return or can take a step. So one of DS6.2, DS6.3, or DS6.8 applies.
   - SS6.4: Rule DS6.6 applies because we can always find an M such that the hypotheses of the rule hold. Specifically, $\text{Dom}(M) = \text{Dom}(V_0)$ where

$\vdash_{\text{v}} s : V_0$ and M maps each x in its domain to a distinct y that is not in V. For such an M, all hypotheses hold.

- SS6.5: Let $e = \mathsf{if}\ e\ s_1\ s_2$. Because e is a well-typed test, Progress Lemma 2 ensures e can take a step or it is some $v \neq \mathsf{junk}$. In the former case, rule DS6.7 applies. In the latter case, either DS6.4 or DS6.5 applies.
- SS6.6: Let $e = \tau\ x$. Rule DS6.1 applies if $x \notin \text{Dom}(H)$. The form of SS6.6 implies $x \notin \text{Dom}(\Gamma_0)$, so $\Gamma_0 \vdash_{\text{htyp}} H : \Gamma_0$ ensures $x \notin \text{Dom}(H)$.
- SS6.7–8: These cases follow from induction.

BIBLIOGRAPHY

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] Alex Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *ACM Conference on Programming Language Design and Implementation*, pages 174–185, La Jolla, CA, June 1995.

[4] Jonathan Aldrich, Craig Chambers, Emin Gün Sirer, and Susan Eggers. Eliminating unnecessary synchronization from Java programs. In *6th International Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pages 19–38, Venice, Italy, September 1999. Springer-Verlag.

[5] Glenn Ammons, Rastislav Bodik, and James Larus. Mining specifications. In *29th ACM Symposium on Principles of Programming Languages*, pages 4–16, Portland, OR, January 2002.

[6] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[7] Andrew Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[8] Andrew Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[9] Andrew Appel. Foundational proof-carrying code. In *16th IEEE Symposium on Logic in Computer Science*, pages 247–258, Boston, MA, June 2001.

[10] Andrew Appel and Amy Felty. A semantic model of types and machine instructions for proof-carrying code. In *27th ACM Symposium on Principles of Programming Languages*, pages 243–253, Boston, MA, January 2000.

[11] J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems*, 20(4):845–868, July 1998.

[12] Todd Austin, Scott Breach, and Gurindar Sohi. Efficient detection of all pointer and array access errors. In *ACM Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, FL, June 1994.

[13] Godmar Back, Wilson Hsieh, and Jay Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *4th USENIX Symposium on Operating System Design and Implementation*, pages 333–346, San Diego, CA, October 2000.

[14] Godmar Back, Patrick Tullmann, Leigh Stoller, Wilson Hsieh, and Jay Lepreau. Techniques for the design of Java operating systems. In *USENIX Annual Technical Conference*, pages 197–210, San Diego, CA, June 2000.

[15] David Bacon, Robert Strom, and Ashis Tarafdar. Guava: A dialect of Java without data races. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 382–400, Minneapolis, MN, October 2000.

[16] Thomas Ball and Sriram Rajamani. Automatically validating temporal safety properties of interfaces. In *8th International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122, Toronto, Canada, May 2001. Springer-Verlag.

[17] Thomas Ball and Sriram Rajamani. The SLAM project: Debugging system software via static analysis. In *29th ACM Symposium on Principles of Programming Languages*, pages 1–3, Portland, OR, January 2002.

[18] Anindya Banerjee and David Naumann. Representation independence, confinement, and access control. In *29th ACM Symposium on Principles of Programming Languages*, pages 166–177, Portland, OR, January 2002.

[19] John Barnes, editor. *Ada 95 Rationale*, volume 1247 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[20] Gregory Bellella, editor. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[21] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In *3rd ACM International Conference on Functional Programming*, pages 129–140, Baltimore, MD, September 1998.

[22] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, safety and performance in the SPIN operating system. In *15th ACM Symposium on Operating System Principles*, pages 267–284, Copper Mountain, CO, December 1995.

[23] Bruno Blanchet. Escape analysis for object oriented languages. Application to Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 20–34, Denver, CO, November 1999.

[24] Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". In *BABEL'01: First International Workshop on Multi-Language Infrastructure and Interoperability*, volume 59(1) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[25] Rastislav Bodk, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bounds checks on demand. In *ACM Conference on Programming Language Design and Implementation*, pages 321–333, Vancouver, Canada, June 2000.

[26] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[27] Herbert Bos and Bart Samwel. Safe kernel programming in the OKE. In *5th IEEE International Conference on Open Architectures and Network Programming*, pages 141–152, New York, NY, June 2002.

[28] Don Box and Chris Sells. *Essential .NET, Volume I: The Common Language Runtime*. Addison-Wesley, 2003.

[29] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230, Seattle, WA, November 2002.

[30] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Safe runtime downcasts with ownership types. Technical Report MIT-LCS-TR-853, Laboratory for Computer Science, MIT, June 2002.

[31] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 56–69, Tampa Bay, FL, October 2001.

[32] John Boyland. Alias burying: Unique variables without destructive reads. *Software Practice and Experience*, 31(6):533–553, May 2001.

[33] Kim Bruce, Luca Cardelli, and Benjamin Pierce. Comparing object encodings. *Information and Computation*, 155:108–133, 1999.

[34] William Bush, Jonathan Pincus, and David Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, June 2000.

[35] David Butenhof. *Programming with POSIX® Threads*. Addison-Wesley, 1997.

[36] *C--*, 2002. `http://www.cminusminus.org`.

[37] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[38] *CCured Documentation*, 2003. `http://manju.cs.berkeley.edu/ccured/`.

[39] *Cforall*, 2002. `http://plg.uwaterloo.ca/~cforall/`.

[40] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, France, 2000. English translation currently available at `http://caml.inria.fr/oreilly-book/`.

[41] Satish Chandra and Tom Reps. Physical type checking for C. In *ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 66–75, Toulouse, France, September 1999.

[42] David Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *ACM Conference on Programming Language Design and Implementation*, pages 296–310, White Plains, NY, June 1990.

[43] Benjamin Chelf, Dawson Engler, and Seth Hallem. How to write system-specific, static checkers in Metal. In *ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 51–60, Charleston, SC, November 2002.

[44] Guang-Ien Cheng, Mingdong Feng, Charles Leiserson, Keith Randall, and Andrew Stark. Detecting data races in Cilk programs that use locks. In *10th ACM Symposium on Parallel Algorithms and Architectures*, pages 298–309, Puerto Vallarta, Mexico, June 1998.

[45] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam Sreedhar, and Sam Midkiff. Escape analysis for Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–19, Denver, CO, November 1999.

[46] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *ACM Conference on Programming Language Design and Implementation*, pages 258–269, Berlin, Germany, June 2002.

[47] Edmund Clarke Jr., Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[48] Christopher Colby, Peter Lee, George Necula, and Fred Blau. A certifying compiler for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 95–107, Vancouver, Canada, June 2000.

[49] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, January 1977.

[50] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium*, pages 63–78, San Antonio, TX, January 1998.

[51] Karl Crary. Toward a foundational typed assembly language. In *30th ACM Symposium on Principles of Programming Languages*, pages 198–212, New Orleans, LA, January 2003.

[52] Cyclone user's manual. Technical Report 2001-1855, Department of Computer Science, Cornell University, November 2001. Current version at `http://www.cs.cornell.edu/projects/cyclone/`.

[53] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A resource accounting interface for Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 21–35, Vancouver, Canada, October 1998.

[54] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *ACM Conference on Programming Language Design and Implementation*, pages 57–68, Berlin, Germany, June 2002.

[55] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, UT, June 2001.

[56] David Detlefs, K. Rustan Leino, Greg Nelson, and James Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[57] Glen Ditchfield. *Contextual Polymorphism*. PhD thesis, University of Waterloo, 1994.

[58] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Detecting memory errors via static pointer analysis (preliminary experience). In *ACM Workshop on Program Analysis for Software Tools and Engineering*, pages 27–34, Montreal, Canada, June 1998.

[59] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Checking cleanness in linked lists. In *7th International Static Analysis Symposium*, volume 1824 of *Lecture Notes in Computer Science*, pages 115–134, Santa Barbara, CA, July 2000. Springer-Verlag.

[60] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *8th International Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 194–212, Paris, France, July 2001. Springer-Verlag.

[61] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *4th USENIX Symposium on Operating System Design and Implementation*, pages 1–16, San Diego, CA, October 2000.

[62] Dawson Engler, David Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *18th ACM Symposium on Operating System Principles*, pages 57–72, Banff, Canada, October 2001.

[63] David Evans. Static detection of dynamic memory errors. In *ACM Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, PA, May 1996.

[64] David Evans, John Guttag, Jim Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *2nd ACM Symposium on the Foundations of Software Engineering*, pages 87–96, New Orleans, LA, December 1994.

[65] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, January 2002.

[66] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *ACM Conference on Programming Language Design and Implementation*, pages 13–24, Berlin, Germany, June 2002.

[67] Manuel Fähndrich and K. Rustan Leino. Non-null types in an object-oriented language. In *ECOOP Workshop on Formal Techniques for Java-like Programs*, June 2002. Published as Technical Report NIII-R0204, Computing Science Department, University of Nijmegen, 2002.

[68] Clive Feather. A formal model of sequence points and related issues, working draft, 2000. Document N925 of ISO/IEC JTC1/SC22/WG14, `http://std.dkuug.dk/JTC1/SC22/WG14/www/docs/n925.html`.

[69] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: a programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.

[70] Kathleen Fisher, Riccardo Pucella, and John Reppy. A framework for interoperability. In *BABEL'01: First International Workshop on Multi-Language Infrastructure and Interoperability*, volume 59(1) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[71] Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, 1997.

[72] Cormac Flanagan and Martín Abadi. Object types against races. In *CONCUR'99—Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 288–303, Eindhoven, The Netherlands, August 1999. Springer-Verlag.

[73] Cormac Flanagan and Martín Abadi. Types for safe locking. In *8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[74] Cormac Flanagan and Stephen Freund. Type-based race detection for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 219–232, Vancouver, Canada, June 2000.

[75] Cormac Flanagan and K. Rustan Leino. Houdini, an annotation assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517, Berlin, Germany, March 2001. Springer-Verlag.

[76] Cormac Flanagan, K. Rustan Leino, Mark Lillibridge, Greg Nelson, James Saxe, and Raymie Stata. Extended static checking for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 2002.

[77] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *ACM International Workshop on Types in Language Design and Implementation*, pages 1–2, New Orleans, LA, January 2003.

[78] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming languages as operating systems (or revenge of the son of the Lisp machine). In *4th ACM International Conference on Functional Programming*, pages 138–147, Paris, France, September 1999.

[79] Matthew Fluet and Riccardo Pucella. Phantom types and subtyping. In *2nd IFIP International Conference on Theoretical Computer Science*, pages 448–460, Montreal, Canada, August 2002. Kluwer.

[80] Jeffrey Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, University of California, Berkeley, 2002.

[81] Jeffrey Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *ACM Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, GA, May 1999.

[82] Jeffrey Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, Germany, June 2002.

[83] Jacques Garrigue and Didier Rémy. Semi-explicit first-class polymorphism for ML. *Information and Computation*, 155(1/2):134–169, 1999.

[84] David Gay. *Memory Management with Explicit Regions*. PhD thesis, University of California, Berkeley, 2001.

[85] David Gay and Alex Aiken. Memory management with explicit regions. In *ACM Conference on Programming Language Design and Implementation*, pages 313–323, Montreal, Canada, June 1998.

[86] David Gay and Alex Aiken. Language support for regions. In *ACM Conference on Programming Language Design and Implementation*, pages 70–80, Snowbird, UT, June 2001.

[87] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1989.

[88] Neal Glew. *Low-Level Type Systems for Modularity and Object-Oriented Constructs*. PhD thesis, Cornell University, 2000.

[89] Neal Glew and Greg Morrisett. Type safe linking and modular assembly language. In *26th ACM Symposium on Principles of Programming Languages*, pages 250–261, San Antonio, TX, January 1999.

[90] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, France, January 1997.

[91] Andrew Gordon and Don Syme. Typing a multi-language intermediate code. In *28th ACM Symposium on Principles of Programming Languages*, pages 248–260, London, England, January 2001.

[92] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[93] Dan Grossman. Existential types for imperative languages: Technical results. Technical Report 2001-1854, Department of Computer Science, Cornell University, October 2001.

[94] Dan Grossman. Existential types for imperative languages. In *11th European Symposium on Programming*, volume 2305 of *Lecture Notes in Computer Science*, pages 21–35, Grenoble, France, April 2002. Springer-Verlag.

[95] Dan Grossman. Type-safe multithreading in Cyclone. In *ACM International Workshop on Types in Language Design and Implementation*, pages 13–25, New Orleans, LA, January 2003.

[96] Dan Grossman and Greg Morrisett. Scalable certification for typed assembly language. In *Workshop on Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 117–145, Montreal, Canada, September 2000. Springer-Verlag.

[97] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *ACM Conference on Programming Language Design and Implementation*, pages 282–293, Berlin, Germany, June 2002.

[98] Dan Grossman, Greg Morrisett, Yanling Wang, Trevor Jim, Michael Hicks, and James Cheney. Formal type soundness for Cyclone's region system. Technical Report 2001-1856, Department of Computer Science, Cornell University, November 2001.

[99] Dan Grossman, Steve Zdancewic, and Greg Morrisett. Syntactic type abstraction. *ACM Transactions on Programming Languages and Systems*, 22(6):1037–1080, November 2000.

[100] Martin Gudgin. *Essential IDL*. Addison-Wesley, 2001.

[101] Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1–4):135–150, 1993.

[102] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific static analyses. In *ACM Conference on Programming Language Design and Implementation*, pages 69–82, Berlin, Germany, June 2002.

[103] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *ACM Conference on Programming Language Design and Implementation*, pages 141–152, Berlin, Germany, June 2002.

[104] Nadeem Hamid, Zhong Shao, Valery Trifonov, Stefan Monnier, and Zhaozhong Ni. A syntactic approach to foundational proof-carrying code. In *17th IEEE Symposium on Logic in Computer Science*, pages 89–100, Copenhagen, Denmark, July 2002.

[105] David Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software Practice and Experience*, 20(1):5–12, January 1990.

[106] Samuel Harbison. *Modula-3*. Prentice-Hall, 1992.

[107] Samuel Harbison and Guy Steele. *C: A Reference Manual, Fifth Edition*. Prentice-Hall, 2002.

[108] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, August 1994.

[109] Robert Harper, Peter Lee, and Frank Pfenning. The Fox project: Advanced language technology for extensible systems. Technical Report CMU-CS-98-107, School of Computer Science, Carnegie Mellon University, January 1998.

[110] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Winter USENIX Conference*, pages 125–138, San Francisco, CA, January 1992.

[111] Chris Hawblitzel and Thorsten von Eicken. Luna: A flexible Java protection system. In *5th USENIX Symposium on Operating System Design and Implementation*, pages 391–403, Boston, MA, December 2002.

[112] Mark Hayden. *The Ensemble System*. PhD thesis, Cornell University, 1998.

[113] Mark Hayden. Distributed communication in ML. *Journal of Functional Programming*, 10(1):91–120, January 2000.

[114] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[115] Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In *ACM International Conference on Principles and Practice of Declarative Programming*, pages 175–186, Florence, Italy, September 2001.

[116] Thomas Henzinger, Ranjit Jhala, Rupak Majumdar, George Necula, Grégoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *14th International Conference on Computer Aided Verification*,

volume 2404 of *Lecture Notes in Computer Science*, pages 526–538, Copenhagen, Denmark, July 2002. Springer-Verlag.

[117] Thomas Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *29th ACM Symposium on Principles of Programming Languages*, pages 58–70, Portland, OR, January 2002.

[118] Michael Hicks, Adithya Nagarajan, and Robbert van Renesse. User-specified adaptive scheduling in a streaming media network. In *6th IEEE International Conference on Open Architectures and Network Programming*, pages 87–96, San Francisco, CA, April 2003.

[119] Gerard Holzmann. Logic verification of ANSI-C code with SPIN. In *7th International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 131–147, Stanford, CA, August 2000. Springer-Verlag.

[120] Gerard Holzmann. Static source code checking for user-defined properties. In *World Conference on Integrated Design and Process Technology*, Pasadena, CA, June 2002. Society for Design and Process Science.

[121] Wilson Hsieh, Marc Fiuczynski, Charles Garrett, Stefan Savage, David Becker, and Brian Bershad. Language support for extensible operating systems. In *Workshop on Compiler Support for System Software*, pages 127–133, Tucson, AZ, February 1996.

[122] Samin Ishtiaq and Peter O'Hearn. BI as an assertion language for mutable data structures. In *28th ACM Symposium on Principles of Programming Languages*, pages 14–26, London, UK, January 2001.

[123] *ISO/IEC 9899:1999, International Standard—Programming Languages—C*. International Standards Organization, 1999.

[124] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *22nd ACM Symposium on Principles of Programming Languages*, pages 393–407, San Francisco, CA, January 1995.

[125] *The Jikes^{TM} Research Virtual Machine User's Guide v2.2.0*, 2003. `http://www.ibm.com/developerworks/oss/jikesrvm`.

[126] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, June 2002.

[127] Stephen Johnson. Lint, a C program checker. Computer Science Technical Report 65, Bell Laboratories, December 1977.

[128] Neil Jones and Steven Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *9th ACM Symposium on Principles of Programming Languages*, pages 66–74, Albuquerque, NM, January 1982.

[129] Richard Jones and Paul Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG'97. Third International Workshop on Automatic Debugging*, volume 2(9) of *Linköping Electronic Articles in Computer and Information Science*, Linköping, Sweden, 1997.

[130] Simon Peyton Jones and John Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language*. `http://www.haskell.org/onlinereport/`, 1999.

[131] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C--: A portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 1–28, Paris, France, September 1999. Springer-Verlag.

[132] Brian Kernighan and Dennis Ritchie. *The C Programming Language, 2nd edition*. Prentice-Hall, 1988.

[133] A. J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993.

[134] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *ACM Conference on Programming Language Design and Implementation*, pages 346–357, Las Vegas, NV, June 1997.

[135] Sumant Kowshik, Dinakar Dhurjati, and Vikram Adve. Ensuring code safety without runtime checks for real-time control systems. In *ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 288–297, Grenoble, France, October 2002.

[136] Dexter Kozen. Efficient code certification. Technical Report 98-1661, Department of Computer Science, Cornell University, January 1998.

[137] Konstantin Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, May 1996.

[138] LCLint user's guide, version 2.5, 2000. `http://splint.org/guide/`.

[139] Xavier Leroy. Unboxed objects and polymorphic typing. In *19th ACM Symposium on Principles of Programming Languages*, pages 177–188, Albuquerque, NM, January 1992.

[140] Xavier Leroy. The effectiveness of type-based unboxing. In *Workshop on Types in Compilation*, Amsterdam, The Netherlands, June 1997. Technical report BCCS-97-03, Boston College, Computer Science Department.

[141] Xavier Leroy. *The Objective Caml system release 3.05, Documentation and user's manual*, 2002. `http://caml.inria.fr/ocaml/htmlman/index.html`.

[142] Sheng Liang. *The Java Native Interface*. Addison-Wesley, 1999.

[143] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[144] Barbara Liskov et al. *CLU Reference Manual*. Springer-Verlag, 1984.

[145] Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Debugging via run-time type checking. In *4th International Conference on Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science*, pages 217–232, Genoa, Italy, April 2001. Springer-Verlag.

[146] QingMing Ma and John Reynolds. Types, abstraction, and parametric polymorphism: Part 2. In *Mathematical Foundations of Programming Semantics*, volume 598 of *Lecture Notes in Computer Science*, pages 1–40, Pittsburgh, PA, March 1991. Springer-Verlag.

[147] Raymond Mak. Sequence point analysis, 2000. Document N926 of ISO/IEC JTC1/SC22/WG14, `http://std.dkuug.dk/JTC1/SC22/WG14/www/docs/n926.html`.

[148] Greg McGary. Bounds checking projects, 2000. `http://www.gnu.org/software/gcc/projects/bp/main.html`.

[149] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[150] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *23rd ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, FL, January 1996.

[151] John Mitchell and Gordon Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[152] *MLton, A Whole Program Optimizing Compiler for Standard ML*, 2002. `http://www.mlton.org`.

[153] Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *ACM Conference on Programming Language Design and Implementation*, pages 81–91, Snowbird, UT, June 2001.

[154] Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, 1995.

[155] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. In *2nd ACM Workshop on Compiler Support for System Software*, pages 25–35, Atlanta, GA, May 1999. Published as INRIA Technical Report 0288, March, 1999.

[156] Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, January 2002.

[157] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[158] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[159] Madanlal Musuvathi, David Park, Andy Chou, Dawson Engler, and David Dill. CMC: A pragmatic approach to model checking real code. In *5th USENIX Symposium on Operating System Design and Implementation*, pages 75–88, Boston, MA, December 2002.

[160] George Necula. Proof-carrying code. In *24th ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.

[161] George Necula. *Compiling With Proofs*. PhD thesis, Carnegie Mellon University, 1998.

[162] George Necula and Peter Lee. The design and implementation of a certifying compiler. In *ACM Conference on Programming Language Design and Implementation*, pages 333–344, Montreal, Canada, June 1998.

[163] George Necula and Peter Lee. Efficient representation and validation of proofs. In *13th IEEE Symposium on Logic in Computer Science*, pages 93–104, Indianapolis, IN, June 1998.

[164] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *29th ACM Symposium on Principles of Programming Languages*, pages 128–139, Portland, OR, January 2002.

[165] George Necula and Shree Rahul. Oracle-based checking of untrusted software. In *28th ACM Symposium on Principles of Programming Languages*, pages 142–154, London, England, January 2001.

[166] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[167] Michael Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.

[168] Michael Norrish. Deterministic expressions in C. In *8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 147–161, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[169] Yunheung Paek, Jay Hoeflinger, and David Padua. Efficient and precise array access analysis. *ACM Transactions on Programming Languages and Systems*, 24(1):65–109, January 2002.

[170] Parveen Patel and Jay Lepreau. Hybrid resource control of active extensions. In *6th IEEE International Conference on Open Architectures and Network Programming*, pages 23–31, San Francisco, CA, April 2003.

[171] Bruce Perens. Electric fence, 1999. `http://www.gnu.org/directory/All_Packages_in_Directory/ElectricFence.html`.

[172] Benjamin Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, 1991.

[173] Benjamin Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM*, 47(3):531–584, 2000.

[174] Benjamin Pierce and David Turner. Local type inference. In *25th ACM Symposium on Principles of Programming Languages*, pages 252–265, San Diego, CA, January 1998.

[175] Willaim Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[176] D. Hugh Redelmeier. Another formalism for sequence points, 2000. Document N927 of ISO/IEC JTC1/SC22/WG14, `http://std.dkuug.dk/JTC1/SC22/WG14/www/docs/n927.html`.

[177] John Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Paris, France, April 1974. Springer-Verlag.

[178] John Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523, Paris, France, September 1983. Elsevier Science Publishers.

[179] Jonathon Rees (eds.) Richard Kelsey, William Clinger. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, September 1998.

[180] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *ACM Conference on Programming Language Design and Implementation*, pages 182–195, Vancouver, Canada, June 2000.

[181] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[182] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[183] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie Mellon University, 1991.

[184] Micahel Siff, Satish Chandra, Thomas Ball, Krishna Kunchithapadam, and Thomas Reps. Coping with type casts in C. In *7th European Software Engineering Conference and 7th ACM Symposium on the Foundations of Software Engineering*, pages 180–198, Toulouse, France, September 1999.

[185] Emin Gün Sirer, Stefan Savage, Przemyslaw Pardyak, Greg DeFouw, Mary Ann Alapat, and Brian Bershad. Writing an operating system using Modula-3. In *Workshop on Compiler Support for System Software*, pages 134–140, Tucson, AZ, February 1996.

[186] Fred Smith, David Walker, and Greg Morrisett. Alias types. In *9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, Berlin, Germany, March 2000. Springer-Verlag.

[187] Geoffrey Smith and Dennis Volpano. Towards an ML-style polymorphic type system for C. In *6th European Symposium on Programming*, volume 1058 of *Lecture Notes in Computer Science*, pages 341–355, Linköping, Sweden, April 1996. Springer-Verlag.

[188] Geoffrey Smith and Dennis Volpano. A sound polymorphic type system for a dialect of C. *Science of Computer Programming*, 32(2–3):49–72, 1998.

[189] Splint manual, version 3.0.6, 2002. `http://www.splint.org/manual/`.

[190] Bjarne Steensgaard. Points-to analysis in almost linear time. In *23rd ACM Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg, FL, January 1996.

[191] Nicholas Sterling. A static date race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, San Diego, CA, January 1993.

[192] Christopher Strachey. Fundamental concepts in programming languages. Unpublished Lecture Notes, Summer School in Computer Programming, August 1967.

[193] Bjarne Stroustrup. *The C++ Programming Language (Special Edition)*. Addison-Wesley, 2000.

[194] S. Tucker Taft and Robert Duff, editors. *Ada 95 Reference Manual*, volume 1246 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[195] David Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML*. PhD thesis, Carnegie Mellon University, 1996.

[196] *The Glasgow Haskell Compiler User's Guide, Version 5.04*, 2002. `http://www.haskell.org/ghc`.

[197] *The Hugs 98 User Manual*, 2002. `http://haskell.cs.yale.edu/hugs`.

[198] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, November 1990.

[199] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):734–767, July 1998.

[200] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions in the ML Kit (for version 4). Technical report, IT University of Copenhagen, September 2001.

[201] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, February 1997.

[202] David Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *7th International Conference on Functional Programming Languages and Computer Architecture*, pages 1–11, La Jolla, CA, June 1995.

[203] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. J-Kernel: A capability-based operating system for Java. In *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[204] Christoph von Praun and Thomas Gross. Object race detection. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–82, Tampa Bay, FL, October 2001.

[205] Philip Wadler. Theorems for free! In *4th International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359, London, England, September 1989. ACM Press.

[206] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.

[207] David Wagner. *Static Analysis and Computer Security: New Techniques for Software Assurance*. PhD thesis, University of California, Berkeley, 2000.

[208] David Wagner, Jeffrey Foster, Eric Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Networking and Distributed System Security Symposium 2000*, pages 3–17, San Diego, CA, February 2000.

[209] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 7(5):203–216, December 1993.

[210] David Walker. *Typed Memory Management*. PhD thesis, Cornell University, 2001.

[211] David Walker, Karl Crary, and Greg Morrisett. Typed memory management in a calculus of capabilities. *ACM Transactions on Programming Languages and Systems*, 24(4):701–771, July 2000.

[212] David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, volume 2071 of *Lecture Notes in Computer Science*, pages 177–206, Montreal, Canada, September 2000. Springer-Verlag.

[213] David Walker and Kevin Watkins. On regions and linear types. In *6th ACM International Conference on Functional Programming*, pages 181–192, Florence, Italy, September 2001.

[214] Daniel Wang and Andrew Appel. Type-preserving garbage collectors. In *28th ACM Symposium on Principles of Programming Languages*, pages 166–178, London, England, January 2001.

[215] Stephanie Weirich. *Programming With Types*. PhD thesis, Cornell University, 2002.

[216] Joe Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, June 1999.

[217] Joe Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A calculus with polymorphic and polyvariant flow types. *Journal of Functional Programming*, 12(3):183–227, May 2002.

[218] Andrew Wright and Robert Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems*, 19(1):87–152, January 1997.

[219] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[220] Writing efficient numerical code in Objective Caml, 2002. `http://caml.inria.fr/ocaml/numerical.html`.

[221] Hongwei Xi. *Dependent Types in Practical Programming.* PhD thesis, Carnegie Mellon University, 1998.

[222] Hongwei Xi. Imperative programming with dependent types. In *15th IEEE Symposium on Logic in Computer Science*, pages 375–387, Santa Barbara, CA, June 2000.

[223] Hongwei Xi and Robert Harper. A dependently typed assembly language. In *6th ACM International Conference on Functional Programming*, pages 169–180, Florence, Italy, September 2001.

[224] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *ACM Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, Canada, June 1998.

[225] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *26th ACM Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, TX, January 1999.

[226] Zhichen Xu. *Safety-Checking of Machine Code.* PhD thesis, University of Wisconsin–Madison, 2001.

[227] Zhichen Xu, Bart Miller, and Tom Reps. Safety checking of machine code. In *ACM Conference on Programming Language Design and Implementation*, pages 70–82, Vancouver, Canada, June 2000.

[228] Zhichen Xu, Tom Reps, and Bart Miller. Typestate checking of machine code. In *10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 335–351, Genoa, Italy, April 2001. Springer-Verlag.

[229] Suan Yong and Susan Horwitz. Reducing the overhead of dynamic analysis. In *2nd Workshop on Runtime Verification*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*, pages 159–179, Copenhagen, Denmark, July 2002. Elsevier Science Publishers.