Type-Safe Multithreading in Cyclone *

Dan Grossman
Computer Science Department
Cornell University
Ithaca, NY 14853
danieljg@cs.cornell.edu

ABSTRACT

We extend Cyclone, a type-safe polymorphic language at the C level of abstraction, with threads and locks. Data races can violate type safety in Cyclone. An extended type system statically guarantees their absence by enforcing that thread-shared data is protected via locking and that thread-local data does not escape the thread that creates it. The extensions interact smoothly with parametric polymorphism and region-based memory management. We present a formal abstract machine that models the need to prevent races, a polymorphic type system for the machine that supports thread-local data, and a corresponding type-safety result.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures, polymorphism*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*operational semantics*

General Terms

Languages

Keywords

data races, types, Cyclone

1. INTRODUCTION

Writing safe and robust low-level code is difficult; writing safe and robust low-level multithreaded code is more difficult. In particular, it takes great care to avoid data races (one thread accessing data while another thread writes the data), but a single race can leave data in an inconsistent state. Because such races might not occur during testing, static systems that prohibit races are valuable.

1.1 Cyclone Needs Race Detection

Cyclone [10, 22] is a type-safe polymorphic programming language that is very close to C. To date, Cyclone has allowed only single-threaded programs, but the domain where Cyclone should prove most useful (low-level and legacy systems with C-style data representation and resource management) has many multithreaded applications. For this reason alone, extending Cyclone's type system to prevent data races would be useful for programmers.

Preventing data races can be even more important: If the target architecture cannot update a source-level pointer atomically, a data race can corrupt a pointer, which violates type safety. If we can translate pointers to machine addresses (as in Cyclone), then this situation arises only if a shared-memory multiprocessor does not ensure atomic writes to memory words.

However, even a system with atomic access for pointers is insufficient because safety can require writing multiple words without an intervening access. For example, Cyclone programmers can define `struct` types where one field holds bounds information for an array that another field points to. Updating such records (to refer to shorter or longer arrays) is an important feature, but we must forbid access while the bounds information does not describe the array.

In short, we have three compelling reasons to use a type system to guarantee the absence of data races:

1. Many programs are not supposed to have races, so static assurances increase reliability.
2. If the updating of references may not be atomic in the implementation, races can corrupt pointers.
3. Type safety can require writes to multiple memory locations before another thread reads any of them.

From the perspective of designing a type-safe language, the first is "optional," but the others are "mandatory." These reasons are not peculiar to Cyclone; they deserve consideration for any safe, low-level, multithreaded language.

1.2 Lock Types to the Rescue

Determining if a multithreaded program has data races is trivially undecidable, but a sound type system can enforce mutual exclusion on access to shared memory. Flanagan, Abadi, and Freund have designed such systems for a small, simply typed, imperative language [14], an object

*This research was supported in part by the AFOSR, under grants F49620-00-1-0198 and F49620-00-1-0209. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of this agency.

TLDI'03, January 18, 2003, New Orleans, Louisiana, USA.

calculus [13], and Java [15]. An implementation of the Java system analyzes large programs, needs only very sparse code annotations, and has found many races. This work inspired multithreaded Cyclone; the rest of this paper explains how we adapt and extend the approach to our language.

Flanagan et al.'s main contribution is a collection of type systems that enforce a discipline in which programmers assign each data object a lock that a thread must hold to access the data. The enabling typing technologies are *singleton lock types*,[1] *quantification over lock names*, and *held-lock effects*.

Section 2 explains these concepts in Cyclone terms. In brief, 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 (hence the term *singleton*). A pointer type's lock name indicates a lock that mediates access to the pointed-to data. A syntax-directed static analysis checks that a thread accesses data only if it holds the appropriate lock. Quantified types let functions be polymorphic over lock names and let data structures abstract over locks that guard their data. Function types include effects describing locks that callers must hold for a function call.

A crucial practical addition 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. Boyapati, Lee, and Rinard's system [4, 5] for race prevention in Java allows lock-name parameters to be instantiated with a special name for thread-local data. That way, one library can let clients pass thread-local or shared data to it.

1.3 Contributions

Our type system makes several contributions beyond applying previous work to a safe C-like language:

1. It is a second-order type system. Cyclone's parametric polymorphism complicates the language for held-lock effects in ways that object types do not. Fortunately, a solution analogous to earlier work integrating polymorphism with explicit memory-management effects [21] is useful, particularly with "caller locks" idioms.

2. 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 requiring only a simple unsynchronized test in the thread-local case.

3. Cyclone threads are safe despite last-in-first-out manual memory management. A combination of static and dynamic checks prevents deallocating memory if any thread might still use it, without always resorting to garbage collection for thread-shared data. This issue is independent of data races, but it is important.

4. A simple kind system (for classifying types, lock names, and memory-region names) collects all of the above additions into a coherent type language.

The technical contribution of this work is a low-level abstract machine, type system, and type-safety proof that capture all of the features described above, except memory management. This formalism includes the first proof for a language with thread-local data. Also, the machine has a dynamic semantics in which type safety requires the absence of data races because mutation takes two steps. An intervening access could make another thread "go wrong." Previous work has prevented races only for abstract machines in which races cannot violate type safety.

The rest of this paper proceeds as follows. In the next section, we describe our basic type system for multithreaded Cyclone. Sections 3 and 4 extend the system by integrating polymorphism and region-based memory management. Section 5 sketches an implementation and necessary run-time support. Section 6 describes our formal system for modeling multithreaded Cyclone and its type safety. Section 7 describes some limitations of our system that future work should address. Section 8 further discusses related work. Section 9 concludes.

We are currently implementing multithreaded Cyclone. A thorough practical evaluation remains future work.

2. RACE-FREE CYCLONE

In this section, we present our extensions for Cyclone multithreading. We assume familiarity with C, focusing on Cyclone's more sophisticated type system. Throughout, we take the liberty of using convenient symbols even though Cyclone has ASCII syntax. Our aim is to:

- 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.

2.1 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`; and executes `f(p2)` in a new thread. The spawned thread terminates when `f` returns; the spawning thread continues execution. After the copy, 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.

Finally, the statement `sync(`e`)` s evaluates e to a lock (or `nonlock`), acquires the lock, executes s, and releases the

[1]The object-oriented systems use a restricted form of dependent type instead, but the idea is similar.

```
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);
}
```

```
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 1: Example. The left side shows a code skeleton with too little type information. It uses some features not in C: `new` e produces a pointer to new memory holding the result of e. `LkInt{.plk=`e_1`,.p=`e_2`}` is a `struct` with fields `plk` and `p` holding the results of e_1 and e_2 respectively. The right side shows a legal Cyclone program (with several optional annotations).

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.

The left side of Figure 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` as 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)`.)

2.2 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 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 follows from 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 quantifiers*, *existential quantifiers*, 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, we can instantiate the `inc` and `inc2` functions in the right side of Figure 1 using any lock name for ℓ. (Section 2.3 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* [23], which is 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 a "current effect" (sometimes called a capability [28]). 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 effect for s. If e has type $\tau*\ell$, then we allow $*e$ only where ℓ is in the current effect. Similarly, a function call type-checks only if the current effect (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 must *unpack* (also called *open*) a value of existential type before using it. In Figure 1, 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.

Existential types are a good example of the need for mutual exclusion, as noted in previous work [19]. 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` while the mutation is in progress (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.

2.3 Multithreading Kinds

We have used several well-known 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` (for "any," for reasons described in Section 3).

In fact, kinds also have *sharabilities*, either `S` (for sharable) or `U` (for maybe unsharable). The lock name for the lock that `newlock` creates has kind `LS` whereas the lock name *loc* has kind `LU`. Kind `LS` is a *subkind* of `LU`, so every lock name can have 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 that 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. The type `sizeof_t<`α`>` is explained in Section 3.

Ensuring thread-local data is unreachable from arguments to spawn is necessary for safety. Making this restriction part of the kind system is a simpler and more uniform approach than *ad hoc* rules that limit the use of `spawn` such that the restriction holds.

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`.

2.4 Default Annotations

The type system we have presented requires a lock name for every pointer type and lock type, and an effect for every function. In practice, simple techniques can make the vast majority of these annotations optional.

First, when a function's effect is omitted, it is filled in with all of the lock names appearing in the parameters' types. In other words, the default is a "caller locks" idiom. Second, lock names are always optional. How they are filled in depends on context:

- Within function bodies, a unification engine can infer lock names.
- For function parameter and return types, we can generate fresh lock names (and include them in default effects). We discuss below options for how many lock names to generate. Top-level functions implicitly universally quantify over free lock names.
- Within type definitions, we use *loc*.

Third, the default sharability for kinds is `U`.

Note that all inference is 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, or might assign different parameters to the same location. Second, we could use *loc* for all omitted lock names. This solution has the advantage that single-threaded programs type-check as multithreaded programs, unless they use global variables. (As Section 7 discusses, global variables require locks.) However, it means that programmers must use extra annotations to write code that is safe for multithreading, even when

callers acquire locks. Because both strategies are 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.

In our example, the first 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.

3. INTEGRATING POLYMORPHISM

Cyclone's *parametric polymorphism* lets a function operate on values of unknown types without the implementation duplicating code or passing types at run-time. The type system is a bit more complicated than in typical polymorphic languages because of nonuniform data representation. To describe locking disciplines for polymorphic code, we need to describe the locks necessary to access a value of unknown type. This section explains the necessary additions.

3.1 Cyclone Polymorphism

In this section, we describe polymorphism (universal quantification over types) for single-threaded Cyclone. The language also has existential quantification over types and type constructors taking type parameters, but polymorphism suffices for explaining the interaction with locks.

This function calls its first argument with its second:

```
void app<α:B>(void f(α), α x) { f(x); }
```

As usual, callers instantiate α with an appropriate type, so the argument to `f` has the type `f` expects. But in Cyclone, the kind `B` (for "boxed") imposes another restriction: α must be instantiated with a pointer type, `int` (in Cyclone, integers and pointers have the same size) or a type variable β of kind `B`. This restriction lets us implement `app` without code duplication or run-time type information, basically because all types of kind `B` have the same size and calling convention.

Type variables with no such restriction have kind `A` (for "any"). Kind `B` is a subkind of `A`. If we change `app` by writing `α:A`, the type of `app` is not well-formed, but we can use kind `A` by adding a level of indirection:

```
void appA<α:A>(void f(α*), α* x) { f(x); }
```

Because programmers control data representation like in C, this distinction between boxed and unboxed types is as natural as the inability to cast a `struct` type to `void*`.

A final extension lets us give types to primitive library routines such as `spawn` that need the size of a value of unknown type: `sizeof_t` is a unary type constructor. The only value of `sizeof_t<τ>` is `sizeof(τ)`, so callers of `spawn` must pass the correct size. As in C, `sizeof(τ)` is ill-formed if the size of τ is not known where it is used.

3.2 Polymorphism and Locks

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 hinted at our solution to the first issue in the previous section: A type's kind includes a sharability (`S` or `U`) in addition to `B`, `A`, or `L`. 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`, and `BS`$\leq$`AU`. Sharability `S` is necessary only for using `spawn`, so almost all code uses sharability `U`.

To extend effects, reconsider the function `app`: Its effect should be the same as the effect for its parameter `f`, but how can we describe `f`'s effect when all we know is that it takes an α of kind `BU`? 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 give `app` this type:

```
void app<α:BU>(void f(α; locks(α)), α x; locks(α));
```

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`)=`$\{\ell_1,\ell_2\}$. As another example, if a polymorphic function calls `app` using β`*`ℓ for α, the current effect 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 2.4, this idiom is what we want if programmers omit explicit effects. Hence the default effect for a polymorphic function includes `locks(α)` for all its type parameters α. In our `app` example, we can omit both effects. In fact, because `B` and `A` are 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 current effect 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. The constraint means, "if ϵ_2 is in the current effect, then it is sound to include ϵ_1 in the current effect."

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 effect (`{`ℓ`}`) and the constraint to cover the effect of `f` (`locks(α)`, which we can

omit). Callers of `app2` must establish the constraint: They must instantiate 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));
```

Instead of `locks(`τ`)`, we could add type-level effect variables, which represent an unknown set of locks much like `locks(`α`)`. As described in previous work [21], introducing effect variables in an explicitly typed language is inconvenient for programmers, especially when using abstract types. The last type we gave `app2` shows that programmers can simulate effect variables with `locks(`τ`)`.

In summary, polymorphism compelled us to add a way to describe the lock names of an unknown type (`locks(`α`)`) and a way to bound 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 it), we need them to use existential types with "caller locks" idioms.

4. INTEGRATING REGIONS

So far, we have described multithreaded Cyclone as if data were never deallocated. Garbage collection can maintain this illusion, but Cyclone's *region-based memory management* gives programmers finer control. In this section, we give a flavor of how the region system is quite analogous to the locking system and how combining the systems allows threads to share reclaimable data.

4.1 Cyclone Regions

We briefly review Cyclone's region system [21]. Each data object is allocated into some region and all the objects in a region are deallocated simultaneously. Regions come in three flavors: There is one *heap region*, which conceptually lives forever. (In practice, it can be garbage collected.) A *stack region* corresponds to a local-declaration block in C. Its lifetime is lexically scoped and allocation into it occurs only when it is created. A *dynamic region* is created via `region r` s. The region is deallocated when control leaves the execution of s. Within s, `r` is bound to the region's *handle*. Dynamic allocation is a primitive that takes a handle. The heap region has a predefined handle.

Handles have types of the form `region_t<`ρ`>` where ρ is a *region name*, i.e., a type variable of kind `R` (for region). The heap's handle has type `region_t<`ρ_H`>`. A region created via `region r` s has type `region_t<`ρ_r`>`. Stack regions do not have handles, but they do have region names, usually generated automatically by the type-checker. Handles exist at run-time whereas region names do not.

Pointer types include the region name of the region into which they point. The type system prevents dereferencing a pointer if its region name indicates the corresponding region may have been deallocated. A function's effect indicates regions that must be live when calling the function.

To describe the regions necessary to access an abstract α, we use the type constructor `regions(`α`)` much as we do `locks(`α`)`. For example, using (unnecessary) explicit effects, the `app` function from the previous section can have the type:

```
void app(void f(α; regions(α), locks(α)), α
         ; regions(α), locks(α));
```

Like lock names, explicit region names are often unnecessary.

The last-in-first-out nature of regions induces subtyping and region bounds: Given `region r1 {...region r2` s`...}`, it is sound to cast from `int*`ρ_{r1} to `int*`ρ_{r2} because `r1` *outlives* `r2`. We write `regions(`τ`):>`ρ to mean every region reachable from a value of type τ outlives ρ. If this bound is true, ρ being live suffices to establish the effect `regions(`τ`)`. Function types can have region-bound preconditions that must hold at call sites.

The correspondence between the static systems for regions and locking is striking. 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, and so on. Regions have subtyping unlike locks because region lifetimes are fixed, but the order of lock acquisitions is not.

4.2 Regions and Locks

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. Although the issues are orthogonal (two objects can have the same region and different locks, or different regions and the same lock), introducing type-level variables that stand for a region name and a lock name can reduce the burden of explicit annotations when this orthogonality is not needed.

The 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`):>`ρ_H. This solution is sound, but it relegates all thread-shared data to the heap.

To add expressiveness, we introduce `rspawn`. We typecheck `rspawn(e1,e2,e3,e4)` like `spawn(e1,e2,e3)` except `e4` has type `region_t<`ρ`>` where the function quantifies over region name ρ and the precondition is `regions(`α`):>`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 described above, 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 point-

ers used in calls to `rspawn`, we enrich region kinds with sharabilities `S` and `U` (as with other kinds), as well as a new 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 "ρ_1 outlives ρ_2," but only if ρ_2 has kind `RD` or ρ_1 has kind `RS`. The handle passed to `rspawn` must be for a region of kind `RS`. Single-threaded Cyclone programs type-check by choosing `RD` for all dynamic-region names.

5. RUN-TIME SYSTEM

The run-time support for Cyclone's basic thread operations is simple. We use the `gcc` C compiler as a back-end. 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 [6]. 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.)

Nonlocal control (jumps, returns, 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 returns, 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 we use sharabilities to restrict region subtyping. Note that if the lists are doubly linked, we add only $O(1)$ amortized cost to `rspawn` and `region`.

6. FORMALISM

We present a formal abstract machine and corresponding type-safety result that models most of the interesting issues regarding Cyclone data-race detection. Omitted details appear in the author's dissertation [20].

The formal machine makes several simplifications in order to focus on safe multithreading. Most importantly:

- All memory is part of a single heap that lives forever. There are no regions or local variables; executing a variable binding allocates a new heap location.
- We do not have types of the form `sizeof_t<`τ`>`. Instead, `spawn` is a statement that can copy a value of any type even though this version of `spawn` would be difficult to implement. For other polymorphism, we are more restrictive than actual Cyclone: We forbid using type variables of kind `AU` or `AS`.
- We use anonymous types, such as product types, instead of named types, such as those defined via `struct`.
- We do not have type constructors.
- We assign and take the address of only whole objects (e.g., `x=`e), not individual fields (e.g., `x.i=`e). Earlier Cyclone formalisms include this orthogonal and cumbersome feature [19, 21].

What remains is still quite powerful. It includes quantified types, effects, constraints, and subkinding as in multi-threaded Cyclone. The semantics for assignment requires *two* rewriting steps for the assigning thread. Between the steps, the assigned-to location holds a "junk" expression, so data races could let threads read junk.

6.1 Syntax

Figure 2 presents the language's syntax. A kind κ includes a sharability σ and whether a type is a lock name (`L`), a boxed type (`B`), or any type except a lock name (`A`). We use τ and ℓ to range over types. Using ℓ is just a convention for types that we know have kind `LU` or `LS`.

Type variables, `int`, and pair types are conventional. A function type includes an effect ϵ describing the locks that callers must hold. Source programs do not have effects containing an integer i. As explained below, we represent locks at run-time with integers and type substitution can produce such effects. If α is not a lock name, then the effect α is like `locks(`α`)` in Cyclone. The pointer type $\tau*\ell$ describes a pointer to an expression of type τ in a location that the lock named ℓ guards. Quantified types (which are α-convertible) can introduce constraints γ, which must hold to instantiate a universal type or introduce an existential type.

A type of the form lock(τ) is like `lock_t<`ℓ`>` in Cyclone. In source programs, we allow only lock(α) or lock(*loc*). During execution, the lock i has type lock($S(i)$) and pointers to locations guarded by lock i have types of the form $\tau*S(i)$.

At the term level, we distinguish statements and expressions, as in C. Most statement forms are like in C with less verbose syntax. These include expressions executed for their effect, return statements, sequential composition, while loops, and conditionals. The variable declaration let x_ℓ=e; s allocates memory. This α-convertible form binds x in s. It allocates fresh memory called x and initialized with the result of e. We can infer the type of x from e. The explicit

$$
\begin{array}{llll}
\text{kinds} & \sigma ::= \mathtt{S} \mid \mathtt{U} & \text{types} & \epsilon ::= \emptyset \mid i \mid \alpha \mid \epsilon \cup \epsilon \\
& \theta ::= \mathtt{B} \mid \mathtt{A} \mid \mathtt{L} & & \gamma ::= \cdot \mid \gamma,\ \epsilon \subseteq \epsilon \\
& \kappa ::= \theta\sigma & & \tau, \ell ::= \alpha \mid \text{int} \mid \tau \times \tau \mid \tau \xrightarrow{\epsilon} \tau \mid \tau{*}\ell \mid \forall\alpha{:}\kappa[\gamma].\tau \mid \exists\alpha{:}\kappa[\gamma].\tau \mid \text{lock}(\ell) \mid loc \mid S(i)
\end{array}
$$

$$
\begin{array}{lll}
\text{terms} & s ::= & e \mid \mathsf{return}\ e \mid s; s \mid \mathsf{while}\ e\ s \mid \mathsf{if}\ e\ s\ s \mid \mathsf{let}\ x_\ell{=}e;\ s \mid \mathsf{open}\ e\ \mathsf{as}\ \alpha, x_\ell;\ s \\
& \mid & \mathsf{spawn}\ e(e) \mid \mathsf{sync}\ e\ s \mid s; \mathsf{release}\ i \\
& e ::= & x \mid i \mid f \mid \&e \mid {*}e \mid (e,e) \mid e.i \mid e{=}e \mid e(e) \mid e[\tau] \mid \mathsf{pack}\ \tau, e\ \mathsf{as}\ \tau \mid \mathsf{nonlock} \mid \mathsf{newlock}() \mid \mathsf{lock}\ i \mid \mathsf{junk}_v \mid \mathsf{call}\ s \\
& f ::= & (\tau\ x_\ell) \xrightarrow{\epsilon} \tau\ s \mid \Lambda\alpha{:}\kappa[\gamma].f
\end{array}
$$

$$
\text{values}\quad v ::= i \mid \&x \mid f \mid (v,v) \mid \mathsf{pack}\ \tau, v\ \mathsf{as}\ \tau \mid \mathsf{nonlock} \mid \mathsf{lock}\ i
$$

$$
\begin{array}{rlll}
\text{heaps} & H & ::= & \cdot \mid H, x \mapsto v \mid H, x \mapsto \mathsf{junk}_v \\
\text{lock sets} & L & ::= & \cdot \mid L, i \\
\text{threads} & T & ::= & L; s \\
\text{program states} & P & ::= & L; L; H; T_1 \ldots T_n
\end{array}
\qquad
\begin{array}{rlll}
\text{static contexts} & \Delta & ::= & \cdot \mid \Delta, \alpha{:}\kappa \\
& \Gamma & ::= & \cdot \mid \Gamma, x{:}(\tau, \ell) \\
& C & ::= & \Delta; \Gamma; \gamma; \epsilon
\end{array}
$$

Figure 2: Syntax of Core Multithreaded Cyclone

lock name ℓ indicates which lock guards the new location. As we explain with the static semantics, it means $\&x$ has some type $\tau{*}\ell$. The form $\mathsf{open}\ e\ \mathsf{as}\ \alpha, x_\ell;\ s$ opens (i.e., unpacks) an existential package. Like the let form, it allocates memory called x and guarded by the lock named ℓ, but the memory holds the contents of the existential package. The scope of the type variable α is s.

The other statement forms provide multithreading and mutual exclusion. $\mathsf{spawn}\ e_1(e_2)$ executes $v_1(v_2)$ in a new thread after evaluating e_1 to v_1 and e_2 to v_2 in the current thread. $\mathsf{sync}\ e\ s$ acquires the lock (or pseudolock) to which e evaluates before executing s. Finally, $s; \mathsf{release}\ i$ does not appear in source programs. It is a place-holder so that the abstract machine releases the lock i after executing s.

Expression forms similar to C include constants (i), variables (x), pointer creation ($\&e$), pointer dereference ($*e$), pair creation ((e,e)), pair projection ($e.i$), assignment ($e{=}e$), and function call ($e(e)$). Type application ($e[\tau]$) and existential creation ($\mathsf{pack}\ \tau, e\ \mathsf{as}\ \tau$) are conventional in polymorphic languages. Functions (f) take a single parameter (which can be a pair) and execute a statement, which must return an expression. Unlike in actual Cyclone, the memory holding the parameter can become thread-shared, so we annotate the formal variable with an ℓ. A function's type abstractions, effect, and assumed constraints are all explicit.

The only remaining source-program forms are nonlock and newlock(), which are a pseudolock for thread-local data and an expression that generates a new lock, respectively. The form $\mathsf{lock}\ i$ is an actual lock. The form junk_v appears when a thread is currently writing v. We use $\mathsf{call}\ s$ to maintain a call stack in the term syntax: A function call is rewritten with this form and the function's return eliminates it.

At run-time, a full machine state $L; L_0; H; T_1 \ldots T_n$ consists of all the locks that have been created (L); the locks currently held by no thread (L_0); a single shared heap (H); and threads (T_i), each of which consists of the locks it holds and a statement, which captures its entire control state. Heaps map variables to expressions. In other words, we reuse variables to serve also as addresses. Most nonvalues are disallowed in the heap, but we need to allow junk_v because mutating a heap location takes two transitions.

We identify heaps and lock sets up to reordering (i.e., they are finite maps and sets respectively), and we write $H_1 H_2$ for combining maps that we implicitly assume have disjoint domains. (Similarly, $L_1 L_2$ is disjoint union).

6.2 Dynamic Semantics

The author's dissertation [20] uses a small-step operational semantics, but here we conserve space by presenting an equivalent contextual semantics. As Figure 3 shows, reduction rules come in three flavors, one each for statement contexts, "left-expression" contexts, and "right-expression" contexts. As in C, a variable is a "left-value" whereas it causes a memory dereference as a right-expression.

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 reduction rules have the form

$$H; (L; L_0; L_h); s \ \rightarrow\ H'; (L'; L_0'; L_h'); s_{\text{opt}}; s'$$

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 locks held.) When the form of (L, L_0, L_h) is unimportant, we abbreviate it as $\overline{L}$.

The most interesting rules are for assignment. Given $x{=}v$, we first change the heap to map x to junk_v and rewrite the expression to $x{=}\mathsf{junk}_v$. The requirement that there is not already junk in x means write-write races can lead to a stuck thread. The subscripted v is necessary so that the next transition remembers what value to finish writing. The semantics allows reading junk_v from the heap, but doing so will likely lead to a stuck thread because there are no destructors for this form. The type system is strong enough that no thread will read junk.

Given the semantics for a single thread, the machine semantics (defined in Figure 4) is straightforward. It is nondeterministic with respect to thread-scheduling. The three rules are for a thread that takes a step and spawns no thread, a thread that takes a step and spawns a thread, and a "cleanup" rule to remove a terminated thread that is holding no locks.

The notation is largely conventional with an important exception. We write $s[\tau/\alpha]$ for the capture-avoiding substitution of τ for α in s (and similarly for expressions, types, and so on). However, the definition for $\epsilon[\tau/\alpha]$ is nonstan-

$$
\begin{array}{rcl}
S & ::= & [\cdot]_S \mid R \mid \mathsf{return}\ R \mid S; s \mid \mathsf{if}\ R\ s_1\ s_2 \mid \mathsf{let}\ x_\ell{=}R;\ s \mid \mathsf{open}\ R\ \mathsf{as}\ \alpha, x_\ell;\ s \\
 & \mid & \mathsf{spawn}\ R(e) \mid \mathsf{spawn}\ v(R) \mid \mathsf{sync}\ R\ s \mid S; \mathsf{release}\ i \\
R & ::= & [\cdot]_R \mid \&L \mid *R \mid (R, e) \mid (v, R) \mid R.i \mid L = e \mid x = R \mid R(e) \mid v(R) \mid R[\tau] \mid \mathsf{pack}\ \tau, R\ \mathsf{as}\ \tau' \mid \mathsf{call}\ S \\
L & ::= & [\cdot]_L \mid *R
\end{array}
$$

$$
\begin{array}{rcl}
H; \overline{L}; (v; s) & \xrightarrow{s} & H; \overline{L}; \cdot; s \\
H; \overline{L}; (\mathsf{return}\ v; s) & \xrightarrow{s} & H; \overline{L}; \cdot; s \\
H; \overline{L}; \mathsf{while}\ e\ s & \xrightarrow{s} & H; \overline{L}; \cdot; \mathsf{if}\ e\ (s; \mathsf{while}\ e\ s)\ 0 \\
H; \overline{L}; \mathsf{if}\ 0\ s_1\ s_2 & \xrightarrow{s} & H; \overline{L}; \cdot; s_2 \\
H; \overline{L}; \mathsf{if}\ i\ s_1\ s_2 & \xrightarrow{s} & H; \overline{L}; \cdot; s_1 \quad (i \neq 0) \\
H; \overline{L}; (\mathsf{let}\ x_\ell{=}v;\ s) & \xrightarrow{s} & H, x \mapsto v; \overline{L}; \cdot; s \quad (x\ \text{fresh}) \\
H; \overline{L}; \mathsf{spawn}\ v_1(v_2) & \xrightarrow{s} & H; \overline{L}; \mathsf{return}\ v_1(v_2); 0 \\
H; \overline{L}; \mathsf{sync\ nonlock}\ s & \xrightarrow{s} & H; \overline{L}; \cdot; s \\
H; (L; L_0, i; L_h); \mathsf{sync\ lock}\ i\ s & \xrightarrow{s} & H; (L; L_0; L_h, i); \cdot; (s; \mathsf{release}\ i) \\
H; (L; L_0; L_h, i); (v; \mathsf{release}\ i) & \xrightarrow{s} & H; (L; L_0, i; L_h); \cdot; v \\
H; (L; L_0; L_h, i); (\mathsf{return}\ v; \mathsf{release}\ i) & \xrightarrow{s} & H; (L; L_0, i; L_h); \cdot; \mathsf{return}\ v \\
H; \overline{L}; (\mathsf{open}\ (\mathsf{pack}\ \tau, v\ \mathsf{as}\ \exists\alpha{:}\kappa[\gamma].\tau')\ \mathsf{as}\ \alpha, x_\ell;\ s) & \xrightarrow{s} & H; \overline{L}; \cdot; (\mathsf{let}\ x_\ell{=}v;\ s[\tau/\alpha]) \\
H; (L; L_0; L_h); \mathsf{newlock}() & \xrightarrow{r} & H; (L, i; L_0, i; L_h); \cdot; \mathsf{pack}\ S(i), \mathsf{lock}\ i\ \mathsf{as}\ \exists\alpha{:}\mathtt{LS}[\cdot].\mathrm{lock}(\alpha) \quad (i\ \text{fresh})
\end{array}
$$

$$
\begin{array}{rcl}
H; \overline{L}; *\&x & \xrightarrow{l} & H; \overline{L}; \cdot; x \\
H; \overline{L}; *\&x & \xrightarrow{r} & H; \overline{L}; \cdot; x \\
H, x \mapsto e; \overline{L}; x & \xrightarrow{r} & H; \overline{L}; \cdot; e \\
H; \overline{L}; (v_0, v_1).i & \xrightarrow{r} & H; \overline{L}; \cdot; v_i \\
H, x \mapsto v'; \overline{L}; x{=}v & \xrightarrow{r} & H, x \mapsto \mathsf{junk}_v; \overline{L}; \cdot; x{=}\mathsf{junk}_v \\
H, x \mapsto \mathsf{junk}_v; \overline{L}; x{=}\mathsf{junk}_v & \xrightarrow{r} & H, x \mapsto v; \overline{L}; \cdot; v \\
H; \overline{L}; ((\tau_1\ x_\ell) \xrightarrow{\epsilon} \tau_2\ s)(v) & \xrightarrow{r} & H; \overline{L}; \cdot; \mathsf{call}\ (\mathsf{let}\ x_\ell{=}v;\ s) \\
H; \overline{L}; (\Lambda\alpha{:}\kappa[\gamma].f)[\tau] & \xrightarrow{r} & H; \overline{L}; \cdot; H, f[\tau/\alpha] \\
H; \overline{L}; \mathsf{call\ return}\ v & \xrightarrow{r} & H; \overline{L}; \cdot; v
\end{array}
$$

$$
\frac{H; \overline{L}; s \xrightarrow{s} H'; \overline{L}'; s_{\mathrm{opt}}; s'}{H; \overline{L}; S[s]_S \rightarrow H'; \overline{L}'; s_{\mathrm{opt}}; S[s']_S}
\qquad
\frac{H; \overline{L}; e \xrightarrow{r} H'; \overline{L}'; s_{\mathrm{opt}}; e'}{H; \overline{L}; S[e]_R \rightarrow H'; \overline{L}'; s_{\mathrm{opt}}; S[e']_R}
\qquad
\frac{H; \overline{L}; e \xrightarrow{l} H'; \overline{L}'; s_{\mathrm{opt}}; e'}{H; \overline{L}; S[e]_L \rightarrow H'; \overline{L}'; s_{\mathrm{opt}}; S[e']_L}
$$

Figure 3: Dynamic Semantics: Single-Thread Transitions

$$
\frac{H; (L; L_0; L_i); s_i \rightarrow H'; (L'; L_0'; L_i'); \cdot; s_i'}{L; L_0; H; T_1 \ldots (L_i; s_i) \ldots T_n \rightarrow L'; L_0'; H'; T_1 \ldots (L_i'; s_i') \ldots T_n}
$$

$$
\frac{H; (L; L_0; L_i); s_i \rightarrow H'; (L'; L_0'; L_i'); s; s_i'}{L; L_0; H; T_1 \ldots (L_i; s_i) \ldots T_n \rightarrow L'; L_0'; H'; T_1 \ldots (L_i'; s_i') \ldots T_n (\cdot; s)}
$$

$$
\frac{}{L; L_0; H; T_1 \ldots T_j (\cdot; \mathsf{return}\ v) T_k \ldots T_n \rightarrow L; L_0; H; T_1 \ldots T_j T_k \ldots T_n}
$$

Figure 4: Dynamic Semantics: Machine Transitions

dard: It means we substitute locks(τ) (defined inductively on the structure of τ [20]) for α. Note that types, and therefore type substitution, have no essential run-time effect.

6.3 Source Static Semantics

We present a sound system for source programs, which are closed statements that do not have terms of the form junk_v, $\mathsf{lock}\ i$, $s; \mathsf{release}\ i$, or $\mathsf{call}\ s$. The next section describes how we extend this system to machine states in order to prove soundness. In the rules and discussion, we omit well-formedness hypotheses and judgments (for example, that an effect must not mention type variables not in scope) that appear in the full definition [20]. Such hypotheses consume space and distract us from the type system's essence. Figure 5 summarizes the judgments used in type-checking.

Figure 6 presents the rules for subkinding ($\vDash_{\mathrm{subk}} \kappa_1 \leq \kappa_2$) and ascribing kinds to types ($\Delta \vDash_{\mathrm{kind}} \tau : \kappa$). Subkinding is quite simple: "boxed" types are "any" types, sharable types are unsharable types, and kinding has a subsumption rule.

The kinding rules are unsurprising. Integers and functions are sharable whereas *loc* is not. Other types have their components' sharability (e.g., a pair is sharable if both components are). Because $\vDash_{\mathrm{subk}} \theta\mathtt{S} \leq \theta\mathtt{U}$, we can require the components to have the same sharability. Only *loc* and α have kinds of the form $\mathtt{L}\sigma$. Integers and pointers are "boxed."

Figure 7 presents the type system, which includes the three interdependent judgments $\vDash_{\mathrm{stmt}}$ (for statements), $\vDash_{\mathrm{rtyp}}$ (for "right-expressions") and $\vDash_{\mathrm{ltyp}}$ (for "left-expressions"). In all cases, the context includes the type variables in scope and their kinds (Δ); the term variables in scope, their types, and the lock names that guard them (Γ); a collection of assumed constraints (γ); and the current effect (ϵ). When convenient, we abbreviate $\Delta; \Gamma; \gamma; \epsilon$ as C and write C_Δ, C_Γ, C_γ, and C_ϵ for the corresponding components of C.

For statements, τ is the return type of the enclosing function; all return statements must have expressions of this type. Statements have only side-effects, so there is no type on the right of the judgment. Right-expressions have a result type as expected. For left-expressions, τ describes the contents of the location and ℓ describes the lock that guards the location. We always describe locations with τ, ℓ whereas we describe right-expressions with just τ.

The type system ensures that the correct lock is held when reading or writing shared data. We write $\gamma; \epsilon \vDash_{\mathrm{acc}} \ell$ to mean, "if the locks described by ϵ are held and the constraints in γ are satisfied, then the lock named ℓ is held." As examples, $\ell \in \epsilon$ suffices, as does $\epsilon = \alpha$ and $\gamma = \ell \subseteq \alpha$. Furthermore, $\gamma; \epsilon \vDash_{\mathrm{acc}} \mathit{loc}$ for all γ and ϵ. Formal rules for this and other omitted judgments are in the author's dissertation [20].

We use $\gamma; \epsilon \vDash_{\mathrm{acc}} \ell$ to derive $C \vDash_{\mathrm{rtyp}} x : \tau$, $C \vDash_{\mathrm{rtyp}} *e : \tau$, and $C \vDash_{\mathrm{rtyp}} e_1{=}e_2 : \tau$ because these are the basic expression forms that could access shared memory. We do not use the judgment for $C \vDash_{\mathrm{ltyp}} e : \tau, \ell$ because there are no left-expression reduction rules that access memory.

We use judgments $\gamma \vDash_{\mathrm{eff}} \epsilon_1 \subseteq \epsilon_2$ and $\gamma_1 \vDash_{\mathrm{eff}} \gamma_2$ for "ϵ_2 must describe a superset of the locks that ϵ_1 describes," and "every constraint in γ_2 is provable from γ_1," respec-

$$\frac{}{\vdash_{\text{subk}} \mathtt{B}\sigma \leq \mathtt{A}\sigma} \qquad \frac{}{\vdash_{\text{subk}} \theta\mathtt{S} \leq \theta\mathtt{U}} \qquad \frac{\vdash_{\text{subk}} \kappa_1 \leq \kappa_3 \quad \vdash_{\text{subk}} \kappa_3 \leq \kappa_2}{\vdash_{\text{subk}} \kappa_1 \leq \kappa_2} \qquad \frac{\Delta \vdash_{\text{kind}} \tau : \kappa \quad \vdash_{\text{subk}} \kappa \leq \kappa'}{\Delta \vdash_{\text{kind}} \tau : \kappa'}$$

$$\frac{}{\Delta \vdash_{\text{kind}} \text{int} : \mathtt{BS}} \qquad \frac{}{\Delta \vdash_{\text{kind}} \mathit{loc} : \mathtt{LU}} \qquad \frac{\Delta(\alpha) \neq \mathtt{AS} \quad \Delta(\alpha) \neq \mathtt{AU}}{\Delta \vdash_{\text{kind}} \alpha : \Delta(\alpha)} \qquad \frac{\Delta, \alpha{:}\kappa \vdash_{\text{kind}} \tau : \kappa' \quad \kappa' \leq \mathtt{AU} \quad \alpha \notin \text{Dom}(\Delta)}{\begin{array}{l}\Delta \vdash_{\text{kind}} \forall\alpha{:}\kappa[\gamma].\tau : \kappa' \\ \Delta \vdash_{\text{kind}} \exists\alpha{:}\kappa[\gamma].\tau : \kappa'\end{array}}$$

$$\frac{\Delta \vdash_{\text{kind}} \ell : \mathtt{L}\sigma}{\Delta \vdash_{\text{kind}} \text{lock}(\ell) : \mathtt{A}\sigma} \qquad \frac{\Delta \vdash_{\text{kind}} \tau_1 : \mathtt{AU} \quad \Delta \vdash_{\text{kind}} \tau_2 : \mathtt{AU}}{\Delta \vdash_{\text{kind}} \tau_1 \xrightarrow{\epsilon} \tau_2 : \mathtt{AS}} \qquad \frac{\Delta \vdash_{\text{kind}} \tau : \mathtt{A}\sigma \quad \Delta \vdash_{\text{kind}} \ell : \mathtt{L}\sigma}{\Delta \vdash_{\text{kind}} \tau * \ell : \mathtt{B}\sigma} \qquad \frac{\Delta \vdash_{\text{kind}} \tau_1 : \mathtt{A}\sigma \quad \Delta \vdash_{\text{kind}} \tau_2 : \mathtt{A}\sigma}{\Delta \vdash_{\text{kind}} \tau_1 \times \tau_2 : \mathtt{A}\sigma}$$

Figure 6: Source Static Semantics: Subkinding and Kinding

$\vdash_{\text{subk}} \kappa_1 \leq \kappa_2$	κ_1 is a subkind of κ_2
$\Delta \vdash_{\text{kind}} \tau : \kappa$	τ has kind κ
$\gamma; \epsilon \vdash_{\text{acc}} \ell$	Given γ and ϵ, the lock named ℓ is held
$\gamma \vdash_{\text{eff}} \epsilon_1 \subseteq \epsilon_2$	Given γ, ϵ_2 provides more access then ϵ_1
$\gamma \vdash_{\text{eff}} \gamma'$	All constraints in γ' are implied by γ
$C \vdash_{\text{ltyp}} e : \tau, \ell$	e is a location of type τ and lock-name ℓ
$C \vdash_{\text{rtyp}} e : \tau$	e has type τ
$C; \tau \vdash_{\text{styp}} s$	s type-checks with return type τ
$\vdash_{\text{ret}} s$	s diverges or becomes some return v (not v)

Figure 5: Source Static Semantics: Judgments

tively. Checking function calls uses the former judgment: the caller must have a current effect that satisfies the effect in the callee's type. Checking existential-package creation and polymorphic-function instantiation uses the latter judgment: the constraints in the quantified types must be satisfied. Dually, to check an open statement or a polymorphic function, we add the constraints to the context.

The only remaining judgment is $\vdash_{\text{ret}} s$. It just describes a conservative analysis to ensure that function bodies do not terminate without executing a return statement.

A few rules merit further discussion. We forbid function bodies from referring to free variables. If allowed, a free reference to a location guarded by *loc* in a function passed to spawn could violate safety. However, we have actually proven type safety for a more relaxed system in which we allow a function body to refer to locations guarded by some lock of kind LS. Note that we type-check function bodies under their explicit effect, not the effect of the context.

The rule for spawn $e_1(e_2)$ requires that the shared value has a sharable kind and that the function is safe for execution in a thread that holds no locks. Finally, the rule for sync e s type-checks s under a stronger current effect, as expected. (Note that locks(loc) $= \emptyset$ whereas locks(α) $= \alpha$.)

6.4 Type Safety

A desirable property would be, "if $\cdot;\cdot;\cdot;\emptyset;\tau \vdash_{\text{styp}} s$ and $\cdot;\cdot;\cdot;(\cdot;s) \rightarrow^* L; L_0; H; T_1 \ldots T_n$ (where $\rightarrow^*$ is the reflexive, transitive closure of $\rightarrow$), then for $1 \leq i \leq n$, either $T_i = (\cdot, \text{return } v)$ for some v (the thread terminated and holds no locks), or $T_i = (L_i; s_i)$ and $H; (L; L_0; L_i); s_i \rightarrow H'; \overline{L}'; s_{\text{opt}}; s_i'$ for some H, $\overline{L}'$, s_{opt}, and s_i'. Informally, no thread becomes stuck.

This property does *not* hold: A thread might be waiting for an unavailable lock, i.e., $s_i =$ sync lock j s and $j \notin L_0$. In fact, nothing prevents deadlocked threads. So we relax our statement of soundness to say every thread has terminated or could take a step *if some additional lock were available.* (A thread may not actually need any additional lock.)

Definition 6.1 (Badly Stuck). *A program $P = (H; L; L_0; T_1 \ldots T_n)$ is badly stuck if it has a badly stuck thread. A badly stuck thread is a thread $(L'; s)$ in P for which there is no v such that $s =$ return v and $L' = \cdot$; and there is no i such that $H; (L; L_0, i; L'); s \rightarrow H'; \overline{L}'; s_{\text{opt}}; s'$ for some H', $\overline{L}'$, s_{opt}, and s'.*

Theorem 6.2 (Type Soundness). *If $\cdot;\cdot;\cdot;\cdot;\emptyset;\tau \vdash_{\text{styp}} s$, $\vdash_{\text{ret}} s$, and $\cdot;(\cdot;\cdot;\cdot);(\cdot;s) \rightarrow^* P$, then P is not badly stuck.*

We use a syntactic proof technique in the style of Wright and Felleisen [29]: We define a typing judgment for machine states $\vdash_{\text{prog}} P$, prove the following lemmas, and conclude Type Soundness as a simple corollary:

- If $\cdot;\cdot;\cdot;\cdot;\emptyset;\tau \vdash_{\text{styp}} s$, then $\vdash_{\text{prog}} \cdot;(\cdot;\cdot;\cdot);(\cdot;s)$.
- If $\vdash_{\text{prog}} P$ and $P \rightarrow P'$, then $\vdash_{\text{prog}} P'$ (or P' has no threads).
- If $\vdash_{\text{prog}} P$, then P is not badly stuck.

The key is getting the right definition for $\vdash_{\text{prog}} P$. We must confront the terms and types not in source programs, the run-time lock sets, and the heap. The rest of this section describes the definition, which is summarized in Figure 8 along with the additional rules and judgments it uses.

First, run-time terms and types can refer to actual locks. These locks should be in L, the set of all created locks. To enforce this restriction, we augment the kinding and typing contexts to include L explicitly. (All rules in Figures 6 and 7 change accordingly.) The resulting rules for the kind of $S(i)$ and the type of lock i are unsurprising. We also need the various lock sets to partition L; i.e., $L = L_0 L_1 \ldots L_n$.

Second, we type-check parts of the heap with the judgment $L; \Gamma \vdash_{\text{htyp}} H : \Gamma'$, which means, "assuming the bindings Γ, heap H provides bindings Γ'." The machine has one shared heap H, but to prove that threads do not badly interfere with each other, we must partition $H = H_S H_{1U} \ldots H_{nU}$. The shared part is H_S: we require $L; \Gamma_S \vdash_{\text{htyp}} H_S : \Gamma_S$ (expressions in H_S cannot refer to other heap locations) and if $x \in \text{Dom}(H_S)$, then the type of $\&x$ has kind AS (it is sharable, i.e., $L \vdash_{\text{shr}} \Gamma_S$). For H_{iU}, we require the addresses of its elements do not have kind AS (they are not sharable, i.e., $L \vdash_{\text{loc}} \Gamma_{iU}$) and the only references to them are

$$\frac{C_\Gamma(x) = \tau, \ell}{C \vdash_{\text{ltyp}} x : \tau, \ell} \qquad \frac{C \vdash_{\text{rtyp}} e : \tau * \ell}{C \vdash_{\text{ltyp}} *e : \tau, \ell} \qquad \frac{C \vdash_{\text{ltyp}} e : \tau, \ell}{C \vdash_{\text{rtyp}} \&e : \tau * \ell} \qquad \frac{C_\Gamma(x) = \tau, \ell \quad C_\gamma; C_\epsilon \vdash_{\text{acc}} \ell}{C \vdash_{\text{rtyp}} x : \tau} \qquad \frac{C \vdash_{\text{rtyp}} e : \tau * \ell \quad C_\gamma; C_\epsilon \vdash_{\text{acc}} \ell}{C \vdash_{\text{rtyp}} *e : \tau}$$

$$\frac{}{C \vdash_{\text{rtyp}} \mathsf{nonlock} : \text{lock}(\mathit{loc})} \qquad \frac{}{C \vdash_{\text{rtyp}} \mathsf{newlock}() : \exists\alpha{:}\mathtt{LS}[\cdot].\text{lock}(\alpha)} \qquad \frac{}{C \vdash_{\text{rtyp}} i : \text{int}}$$

$$\frac{C \vdash_{\text{rtyp}} e : \tau[\tau_1/\alpha] \quad C_\Delta \vdash_{\text{kind}} \tau_1 : \kappa \quad C_\gamma \vdash_{\text{eff}} \gamma'[\tau_1/\alpha] \quad C_\Delta \vdash_{\text{kind}} \exists\alpha{:}\kappa[\gamma'].\tau : \mathtt{AU}}{C \vdash_{\text{rtyp}} \mathsf{pack}\ \tau_1, e\ \mathsf{as}\ \exists\alpha{:}\kappa[\gamma'].\tau : \exists\alpha{:}\kappa[\gamma'].\tau} \qquad \frac{\Delta; x{:}(\tau_1, \ell); \gamma; \epsilon'; \tau_2 \vdash_{\text{styp}} s \quad \vdash_{\text{ret}} s}{\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} (\tau_1\ x_\ell) \xrightarrow{\epsilon'} \tau_2\ s : \tau_1 \xrightarrow{\epsilon'} \tau_2}$$

$$\frac{\Delta, \alpha{:}\kappa; \Gamma; \gamma, \gamma'; \epsilon \vdash_{\text{rtyp}} f : \tau}{\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} \Lambda\alpha{:}\kappa[\gamma'].f : \forall\alpha{:}\kappa[\gamma'].\tau} \qquad \frac{C \vdash_{\text{rtyp}} e : \forall\alpha{:}\kappa[\gamma'].\tau' \quad C_\Delta \vdash_{\text{kind}} \tau : \kappa \quad C_\gamma \vdash_{\text{eff}} \gamma'[\tau/\alpha]}{C \vdash_{\text{rtyp}} e[\tau] : \tau'[\tau/\alpha]} \qquad \frac{C \vdash_{\text{rtyp}} e_1 : \tau_1 \quad C \vdash_{\text{rtyp}} e_2 : \tau_2}{C \vdash_{\text{rtyp}} (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{C \vdash_{\text{ltyp}} e_1 : \tau, \ell \quad C \vdash_{\text{rtyp}} e_2 : \tau \quad C_\gamma; C_\epsilon \vdash_{\text{acc}} \ell}{C \vdash_{\text{rtyp}} e_1{=}e_2 : \tau} \qquad \frac{C \vdash_{\text{rtyp}} e_1 : \tau_1 \xrightarrow{\epsilon'} \tau_2 \quad C \vdash_{\text{rtyp}} e_2 : \tau_1 \quad C_\gamma \vdash_{\text{eff}} \epsilon' \subseteq \epsilon}{C \vdash_{\text{rtyp}} e_1(e_2) : \tau_2} \qquad \frac{C \vdash_{\text{rtyp}} e : \tau_0 \times \tau_1 \ \ i{\in}\{0,1\}}{C \vdash_{\text{rtyp}} e.i : \tau_i}$$

$$\frac{C \vdash_{\text{rtyp}} e : \tau'}{C; \tau \vdash_{\text{styp}} e} \qquad \frac{C \vdash_{\text{rtyp}} e : \tau}{C; \tau \vdash_{\text{styp}} \mathsf{return}\ e} \qquad \frac{C; \tau \vdash_{\text{styp}} s_1 \quad C; \tau \vdash_{\text{styp}} s_2}{C; \tau \vdash_{\text{styp}} s_1; s_2} \qquad \frac{C \vdash_{\text{rtyp}} e : \text{int} \quad C; \tau \vdash_{\text{styp}} s_1 \quad C; \tau \vdash_{\text{styp}} s_2}{C; \tau \vdash_{\text{styp}} \mathsf{if}\ e\ s_1\ s_2}$$

$$\frac{C \vdash_{\text{rtyp}} e : \text{int} \quad C; \tau \vdash_{\text{styp}} s}{C; \tau \vdash_{\text{styp}} \mathsf{while}\ e\ s} \qquad \frac{x \notin \text{Dom}(\Gamma) \quad \Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} e : \tau' \quad \Delta; \Gamma, x{:}(\tau', \ell); \gamma; \epsilon; \tau \vdash_{\text{styp}} s}{\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{\text{styp}} \mathsf{let}\ x_\ell{=}e;\ s}$$

$$\frac{\alpha \notin \text{Dom}(\Delta) \quad x \notin \text{Dom}(\Gamma) \quad \Delta \vdash_{\text{kind}} \ell : \mathtt{LU} \quad \Delta \vdash_{\text{kind}} \tau : \mathtt{AU} \quad \Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} e : \exists\alpha{:}\kappa[\gamma'].\tau' \quad \Delta, \alpha{:}\kappa; \Gamma, x{:}(\tau', \ell); \gamma, \gamma'; \epsilon; \tau \vdash_{\text{styp}} s}{\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{\text{styp}} \mathsf{open}\ e\ \mathsf{as}\ \alpha, x_\ell;\ s}$$

$$\frac{C \vdash_{\text{rtyp}} e_1 : \tau_1 \xrightarrow{\emptyset} \tau_2 \quad C \vdash_{\text{rtyp}} e_2 : \tau_1 \quad C_\Delta \vdash_{\text{kind}} \tau_1 : \mathtt{AS}}{C; \tau \vdash_{\text{styp}} \mathsf{spawn}\ e_1(e_2)} \qquad \frac{\Delta; \Gamma; \gamma; \epsilon \vdash_{\text{rtyp}} e : \text{lock}(\ell) \quad \Delta; \Gamma; \gamma; \epsilon \cup \text{locks}(\ell); \tau \vdash_{\text{styp}} s}{\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{\text{styp}} \mathsf{sync}\ e\ s}$$

Figure 7: Source Static Semantics: Typing

from s_i and H_{iU}. But they can refer to H_S, so we require $L; \Gamma_S\Gamma_{iU} \vdash_{\text{htyp}} H_{iU} : \Gamma_{iU}$. This partition provides a strong enough induction hypothesis to establish that there is never a race on a location guarded by *loc*.

Third, if $L_i = i_1, \ldots, i_n$, it is too weak to type-check s_i in a context with $\epsilon = i_1 \cup \cdots \cup i_n$: If s_i releases lock i_j and becomes s_i', we must know s_i' does not need i_j to type-check. The solution is to use $\epsilon = \emptyset$ (so $L; \cdot; \Gamma_S\Gamma_{iU}; \cdot; \emptyset; \tau \vdash_{\text{styp}} s_i$) and have the typing rule for $s; \mathsf{release}\ i_j$ add i_j to ϵ for checking s. But we still do not enforce that (terminating) threads release exactly the locks they hold and return values. For the latter, $\vdash_{\text{ret}} s_i$ suffices. For the former, we introduce the judgment $L \vdash_{\text{srel}} s$. Informally, $L \vdash_{\text{srel}} s$ holds when the release statements in s mention exactly the locks in L exactly once each and all release statements are "between" the active redex and the root of s (viewed as an abstract-syntax tree). (The formal definition is simple and syntax-directed [20].)

Finally, the typing rule for junk_v does not restrict where such terms appear. Instead, we use the $\vdash_{\text{j}} H; s$ judgment to impose a strong invariant about junk: We forbid junk_v everywhere except s_i can be $S[x{=}\mathsf{junk}_v]$ (for some x and evaluation context S), in which case we require $H(x) = \mathsf{junk}_v$. So there is at most one junk location for each thread. Furthermore, x must be in H_{iU} (the thread's local heap) or H_{iS}, the part of H_S guarded by a lock the thread holds (as enforced with $\Gamma_S; L_i \vdash_{\text{hlk}} H_{iS}$). In either case ($x \in \text{Dom}(H_{iU})$ or $x \in \text{Dom}(H_{iS})$), no other thread will access x before thread i takes another step. Similarly, if s_i has the form $S[x{=}v]$, then thread i and only thread i can access x, so x cannot contain junk.

$$\frac{i \in L}{L; \Delta \vdash_{\text{kind}} S(i) : \mathtt{LS}} \qquad \frac{i \in C_L}{C \vdash_{\text{rtyp}} \mathsf{lock}\ i : \text{lock}(S(i))}$$

$$\frac{C \vdash_{\text{rtyp}} v : \tau}{C \vdash_{\text{rtyp}} \mathsf{junk}_v : \tau} \qquad \frac{C; \tau \vdash_{\text{styp}} s \quad \vdash_{\text{ret}} s}{C \vdash_{\text{rtyp}} \mathsf{call}\ s : \tau}$$

$$\frac{i \in L \quad L; \Delta; \Gamma; \gamma; \epsilon \cup i; \tau \vdash_{\text{styp}} s}{L; \Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{\text{styp}} s; \mathsf{release}\ i}$$

$L; \Gamma \vdash_{\text{htyp}} H : \Gamma'$	Heap H has type Γ'
$\Gamma; L \vdash_{\text{hlk}} H$	All $x \in \text{Dom}(H)$ are locked by an $i \in L$
$L \vdash_{\text{shr}} \Gamma$	All $x \in \text{Dom}(\Gamma)$ are sharable
$L \vdash_{\text{loc}} \Gamma$	No $x \in \text{Dom}(\Gamma)$ are sharable
$L \vdash_{\text{srel}} s$	s releases exactly L (or diverges)
$\vdash_{\text{j}} H; s$	H and s are junk-free or $H = H', x \mapsto \mathsf{junk}_v$ and $s = S[x{=}\mathsf{junk}_v]$ where H' and S are junk-free

$$\frac{\begin{array}{l} H = H_S H_{1U} \ldots H_{nU} \qquad L = L_0 L_1 \ldots L_n \\ H_S = H_{0S} H_{1S} \ldots H_{nS} \\ L; \Gamma_S \vdash_{\text{htyp}} H_S : \Gamma_S \quad L \vdash_{\text{shr}} \Gamma_S \quad \Gamma_S; L_0 \vdash_{\text{hlk}} H_{0S} \quad \vdash_{\text{j}} H_{0S}; 0 \\ \text{for all } 1 \le i \le n \\ L; \Gamma_S\Gamma_{iU} \vdash_{\text{htyp}} H_{iU} : \Gamma_{iU} \quad L \vdash_{\text{loc}} \Gamma_{iU} \quad \Gamma_S; L_i \vdash_{\text{hlk}} H_{iS} \\ L; \cdot; \Gamma_S\Gamma_{iU}; \cdot; \emptyset; \tau_i \vdash_{\text{styp}} s_i \quad \vdash_{\text{ret}} s_i \quad L_i \vdash_{\text{srel}} s_i \quad \vdash_{\text{j}} H_{iS} H_{iU}; s_i \end{array}}{\vdash_{\text{prog}} L; L_0; H; (L_1; s_1) \ldots (L_n; s_n)}$$

Figure 8: Program-State Typing ($C ::= L; \Delta; \Gamma; \gamma; \epsilon$)

7. LIMITATIONS

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 never mutated does not need locking. Expressing this read-only invariant is straightforward if we "take `const` seriously" (i.e., prevent mutation of `const` data, unlike C), but *qualifier polymorphism* [18] 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.

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 dataflow analysis will probably suffice to allow access without locking so long as an object could not yet have become shared.

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. Cyclone's designers are exploring ways to allow some safe uses of memory deallocation (`free`). The issues are so similar (deallocation must not precede access) that techniques for safe deallocation should also support object migration.

Finally, 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.

8. RELATED WORK

As discussed in Section 1, this work is closely related to the static race-prevention systems developed by Flanagan et al. [14, 13, 15] and Boyapati et al. [5, 4]. On the surface, the main difference is that these systems have targeted Java, so they need not interact with parametric polymorphism or memory management. Also, Java programmers enjoy the convenience of every object being a lock. The tricky issue for Java is run-time downcasts: A sound system must check that a cast from `Object` to a subclass `Foo` with a field `f` guarded by a lock ℓ is correct even though `Foo` is parameterized by the lock for `f`. Recent systems [4] have resorted to run-time type passing whereas Cyclone enjoys type erasure. Boyapati et al.'s system supports read-only data and object migration much as described in the previous section. These advanced systems have been implemented and evaluated on real applications, but they lack type-safety proofs.

Guava [2] 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.

Sacrificing soundness (potentially missing data races) can reduce false positives and explicit annotations. The results can be very useful for nonmalicious code. Examples include ESC/Java [17], which usually acts as though loops iterate only once, and Warlock [26], a static race-detector for C that makes optimistic aliasing assumptions and assumes locations holding locks are not mutated to hold different locks.

There are many other race-detection systems, some of which are dynamic [7, 9, 25, 27]. 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.

There are race-free disciplines other than what Cyclone's type system enforces, of course. A more flexible system could let programs specify one via a verification condition. Flanagan, Freund, and Qadeer [16] explain how to verify specifications in a thread-modular fashion. Adding such flexibility to Cyclone would require preventing specifications that permit data races.

Static analyses that find thread-local data can eliminate unnecessary locking in Java [1, 3, 8]. Adapting such interprocedural escape analyses to Cyclone would reduce annotations but complicate the language definition.

Other work on safe languages for low-level applications has not allowed threads. In Vault [11, 12], 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 [24], unmodified legacy C applications are compiled unconventionally (with extra data fields and run-time checks) to detect memory-safety violations. The key to performance is a whole-program static analysis to eliminate many unnecessary fields and 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.

9. CONCLUSION

We have presented a type system that prevents data races in a multithreaded extension of Cyclone that includes parametric polymorphism and region-based memory management. The programmer must use locks for thread-shared data, but not for thread-local data. A formal abstract machine models the system's key features, including thread-local data. We have a rigorous proof of type safety, which for our abstract machine implies that there are no data races.

Acknowledgments

Kevin O'Neill, Yanling Wang, Stephanie Weirich, and the anonymous reviewers provided valuable feedback that improved this work.

10. REFERENCES

[1] J. Aldrich, C. Chambers, E. G. Sirer, and S. 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, Sept. 1999. Springer-Verlag.

[2] D. Bacon, R. Strom, and A. 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, Oct. 2000.

[3] B. 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, Nov. 1999.

[4] C. Boyapati, R. Lee, and M. 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, Nov. 2002.

[5] C. Boyapati and M. 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, Oct. 2001.

[6] D. Butenhof. *Programming with POSIX® Threads*. Addison-Wesley, 1997.

[7] G.-I. Cheng, M. Feng, C. Leiserson, K. Randall, and A. 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.

[8] J.-D. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–19, Denver, CO, Nov. 1999.

[9] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. 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.

[10] *Cyclone User's Manual*, 2002. `http://www.cs.cornell.edu/projects/cyclone/`.

[11] R. DeLine and M. 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.

[12] M. Fähndrich and R. 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.

[13] C. Flanagan and M. Abadi. Object types against races. In *CONCUR'99—Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 288–303, Eindhoven, The Netherlands, Aug. 1999. Springer-Verlag.

[14] C. Flanagan and M. 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, Mar. 1999. Springer-Verlag.

[15] C. Flanagan and S. Freund. Type-based race detection for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 219–232, Vancouver, Canada, June 2000.

[16] C. Flanagan, S. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *11th European Symposium on Programming*, volume 2305 of *Lecture Notes in Computer Science*, pages 262–277, Grenoble, France, Apr. 2002. Springer-Verlag.

[17] C. Flanagan, K. R. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 2002.

[18] J. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *ACM Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, GA, May 1999.

[19] D. 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, Apr. 2002. Springer-Verlag.

[20] D. Grossman. *Safe Programming at the C Level of Abstraction*. PhD thesis, Cornell University, 2003. Forthcoming.

[21] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *ACM Conference on Programming Language Design and Implementation*, pages 282–293, Berlin, Germany, June 2002.

[22] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, June 2002.

[23] J. Lucassen and D. Gifford. Polymorphic effect systems. In *15th ACM Symposium on Principles of Programming Languages*, pages 47–57, San Diego, CA, Jan. 1988.

[24] G. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *29th ACM Symposium on Principles of Programming Languages*, pages 128–139, Portland, OR, Jan. 2002.

[25] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.

[26] N. Sterling. A static date race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.

[27] C. von Praun and T. Gross. Object race detection. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–82, Tampa Bay, FL, Oct. 2001.

[28] D. Walker, K. Crary, and G. Morrisett. Typed memory management in a calculus of capabilities. *ACM Transactions on Programming Languages and Systems*, 24(4):701–771, July 2000.

[29] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.