Safe and Flexible Memory Management in Cyclone

Michael Hicks
University of Maryland
mwh@cs.umd.edu

Greg Morrisett Dan Grossman
Cornell University
{jgm,danieljg}@cs.cornell.edu

Trevor Jim
AT&T Labs Research
trevor@research.att.com

ABSTRACT

Cyclone is a type-safe programming language intended for applications requiring control over memory management. Our previous work on Cyclone included support for stack allocation, lexical region allocation, and a garbage-collected heap. We achieved safety (i.e., prevented dangling pointers) through a region-based type-and-effects system. This paper describes some new memory-management mechanisms that we have integrated into Cyclone: dynamic regions, unique pointers, and reference-counted objects. Our experience shows that these new mechanisms are well suited for the timely recovery of objects in situations where it is awkward to use lexical regions. Crucially, programmers can write reusable functions without unnecessarily restricting callers' choices among the variety of memory-management options. To achieve this goal, Cyclone employs a combination of polymorphism and scoped constructs that temporarily let us treat objects as if they were allocated in a lexical region.

1. INTRODUCTION

Cyclone is a type-safe, C-like language intended for use in systems programs where control is needed over low-level details such as data representations and resource management. In previous work [22], we described a region-based type system for Cyclone, based on the work of Tofte and Talpin [39], that gives programmers type-safe support for stack allocation and lexically-scoped regions. We also showed how these manual memory-management mechanisms could be safely combined with heap allocation and a (conservative) garbage collector to give programmers a range of memory-management options. One attractive feature of the design is that all data objects are treated as if they live in some region. Using region *polymorphism*, one can write library routines accepting pointers into any part of memory, including the stack, a lexical region, or the heap.

Stack allocation is an important and pervasive idiom in C programs, providing efficient allocation, access, and deallocation. Region allocation is another important memory-management idiom, due to the efficiency of batched deallocation of objects. It is used in compilers such as LCC [18] and servers such as Apache [4]. However, Cyclone's type system previously supported only regions that followed a strict last-in-first-out (LIFO) discipline. The LIFO restriction keeps type checking simple and naturally provides a form of region subtyping that is important for writing reusable code (see Section 2).

Unfortunately, LIFO behavior has well-known limitations. A key problem is that an object's lifetime is fixed when it is allocated, so subsequent input and computation can neither shorten nor extend the lifetime. Furthermore, regions are efficient for large collections of objects that need to be deallocated together, but they are less so for small collections. In particular, the costs for creating and deallocating a region make regions expensive for single objects. Until now, garbage collection has been our best solution in such cases.

The work described here augments Cyclone with additional type-safe memory-management options to help programmers with these situations. The new options, described in Sections 3–5, include (a) dynamic regions, (b) unique pointers, and (c) reference-counted objects. *Dynamic regions* provide support for region deallocation at almost any program point and thus can be used to avoid the LIFO constraints of lexical regions. However, this flexibility incurs some run-time overhead and possible exceptions. *Unique pointers* are based on linear (more properly, affine) type systems and provide lightweight memory management for individual objects. In particular, a unique pointer's object can be deallocated at any program point. However, unique pointers cannot be freely copied, and there are restrictions on how they can be accessed when placed in a shared object. Finally, pointers to *reference-counted objects* are treated similarly to unique pointers except that copies of the pointer are allowed at the price of maintaining a reference count. When all copies of the pointer are destroyed, the object is deallocated.

In our experience, it is extremely useful to have such a large set of memory-management options so that programmers can choose a strategy that works best for their application. Section 6 demonstrates how we tuned the performance of two systems applications: an event-based web server and MediaNet [29], an overlay network for streaming data. In both cases we were able to keep memory consumption very low. For MediaNet, using unique and reference-counted pointers increased throughput by up to 42% compared to relying entirely on conservative garbage collection.

However, there is a danger that so many different options will overly complicate the language and make it impossible to write reusable libraries. Thus, our most important contribution is a design that focuses on uniformity and code reuse. For example, dynamic regions reuse the lexical region machinery, and reference-counts are built on top of unique pointers. Furthermore, we provide constructs called `open` and `alias` that support controlled "pinning" for dynamic regions and controlled aliasing for unique pointers respectively. Crucially, these constructs let programmers write generic functions that can operate over lexical-region, dynamic-region, unique or reference-counted pointers.

2. LEXICAL REGIONS

We begin by reviewing our previous work on region-based memory management for Cyclone [22]. We then describe important limitations of this work.

2.1 Review

All memory objects in Cyclone are placed in a logical container called a region The previous version of Cyclone had three basic kinds of regions: There is one *heap region* (`‘H`) with global scope that conceptually lives forever. Objects allocated in the heap cannot be reclaimed except with an optional conservative garbage collector.

Stack regions correspond to local-declaration blocks. Entering a block creates a stack region and allocates space in that region for the local variables. When control exits the block, the stack region's objects are deallocated. For example, in the function:

```
void foo(int x) {
  if (x)
    L:{ int y = 3;
        bar(&y);
      }
}
```

entering the block labeled `L` creates a region named `‘L` and allocates space for the local variable `y`. The region is deallocated after the call to `bar`. If the programmer omits the label on a block, the compiler generates a fresh label. Stack regions are really a special case of lexical regions that admits a faster implementation by disallowing dynamic allocation.

Lexical regions[1] also have creation and deallocation determined by scope, but a handle lets the program allocate objects into the region throughout the region's lifetime. Allocation primitives take handles so programs determine object lifetimes at an allocation site. For example, in the function:

```
void baz() {
  { region<‘r> h;
    int *x = rmalloc(h,sizeof(int));
    *x = 3;
    int *y = baf(h,x);
  }
}
```

we create a fresh region named `‘r` with an allocation handle `h`. The handle can be passed to `rmalloc` to allocate storage. It can also be passed to a user-defined function, such as `baf`, so a callee can allocate data in the region, and return results that might point into it. In our example, all data placed in `h`'s region is deallocated after the call to `baf`. The ability to pass handles as first-class objects lets us allocate a dynamically-determined number of objects in any caller's region.

The primary goal of the type system is to ensure programs never dereference dangling pointers. To do so, we track the set of regions that are live at each program point, and augment pointer types with the *region name* of the region into which the value points. Thus, an attempt to dereference a pointer into a region is allowed only if the region is still live. The lexical scoping discipline makes it easy to track the set of live regions statically because deallocation happens only at structured program points.

Region names are type-level variables that describe regions instead of types. For example, `‘H` is the region name for the heap, and `int *‘H` is a heap pointer. Lexical regions have names that are in scope for the corresponding code block. Handles have types of the form `region_t<‘r>`, where `‘r` is the name of the region into which objects are placed when the handle is used for allocation. For instance, if `h` has type `region_t<‘r>`, then `rmalloc(h,sizeof(int))` returns a pointer of type `int *‘r`. We use intraprocedural type inference and well-chosen defaults to avoid writing many region annotations; for example, the region annotations on `x`, `y`, and `h` are inferred in the examples above.

Functions and type constructors may be parameterized by type and region variables. For example, the following `length` function accepts lists with any element type and with the list spine allocated in any (still-live) region.

```
struct List<‘a,‘r> {
  ‘a hd;
  struct List<‘a,‘r> *‘r tl;
};
int length(struct List<‘a,‘r>*‘r lst) {
  int i=0;
  for(; lst != NULL; lst = lst->tl) ++i;
  return i;
}
```

For safety, a pointer type is considered well formed only when its region name is in scope. For example, consider a function that tries to return a dangling pointer to a local variable:

```
int *‘L bad() {
 L: { int x = 3;
      return &x;
     }
}
```

Because `x` is declared in block `L`, the address of `x` has type `int *‘L`. Our scoping rules state that `‘L` is not in scope outside the block, so `int *‘L` is not well-formed as a return type, and Cyclone would flag this as an error.

However, Cyclone supports *existential types*, which can hide a region in a function's return type. For example, one can write something similar to:

```
(∃‘r. int *‘r) bad() {
  L: { int x = 3;
       return pack(‘L,&x) as ∃‘r. int *‘r;
     }
}
```

[1] Our previous paper [22] referred to lexical regions as *dynamic regions* due to their dynamically-determined sizes; in this paper we use the latter term for regions with dynamically-determined lifetimes.

(The actual syntax for existentials is shown in Section 6). The result type is well-formed and allows a dangling pointer. Thus, in general, the set of live regions is a subset of those that are in scope. To prevent access to a deallocated region, the type system keeps track of which regions are live at each program point. An intraprocedural analysis is extended across function boundaries by requiring an explicit *effect* that records the set of regions that must be live across the call. By default, Cyclone assumes all region parameters are live across the call. In practice, this default works well and thus programmers almost never write explicit effects.

Finally, we have a natural notion of subtyping: If the region named ‘r1 *outlives* the region named ‘r2, then we can coerce a value of type τ *‘r1 to type τ *‘r2 because the latter type allows access at strictly fewer program points. For instance, the following code is well-formed:

```
void h(int *‘r1 x, int *‘r2 y) {
  L:{ int *‘L z = (rand()) ? x : y;
       ...
    }
}
```

Note that z is assigned either a ‘r1 or ‘r2 pointer. Since both regions must be live across the call, they naturally outlive ‘L, so we can safely promote x and y to ‘L pointer types. If regions did not have structured scope, such subtyping would not arise naturally. We remark that the type system supports *deep* subtyping along read-only pointers. Thus, if τ_1 is a subtype of τ_2 and ‘r1 outlives ‘r2, then τ_1* ‘r1 is a subtype of τ_2 const *‘r2.

For this and several other reasons, lexical regions lead to convenient programming and a simple type system. Perhaps the most compelling advantage is that the system is completely static, so there is no need for run-time checks.

2.2 Limitations of Lexical Regions

Unfortunately, lexical regions provide insufficient control over memory lifetimes. First, the region-deallocation point is determined at region-allocation time, so programs cannot choose to deallocate based on computation following region allocation. For instance, we cannot decide to deallocate a region based on a user input.

Second, regions are often forced to live longer than necessary. For example, a callee cannot deallocate a region allocated by a caller, even if the caller will not later access the region. Conversely, callees cannot give callers freshly allocated regions, which forces callers to allocate regions earlier than necessary. This restriction makes important idioms impossible, such as the copying collector of Wang and Appel [44]. In general, any iterative process that maintains state across iterations is forced to leak memory; the state must reside in a region allocated outside of the loop. In other words, there is no support for a “tail-call” that deallocates a region before performing a call.

Third, objects often live longer than necessary because pointers allocated before a region exists cannot be used to access an object in the region. In particular, global variables can access only heap-allocated data.

Fourth, manipulating nonstack regions takes more time and space than using malloc/free for an individual object. For regions holding many objects, amortization overcomes this cost. But for many programs, individual objects have distinct points of “last use” so aggregating lifetimes retains excessive memory.

In other implementations and designs, these limitations have been noted and partially addressed. For instance, the ML Kit compiler [38] includes a special reset primitive that is used to deallocate regions early, but its use is an internal optimization whose soundness is not captured by the type system. The Capability Calculus [41] supports deallocation at any program point, but requires much more elaborate effects and region aliasing information. Other approaches are discussed in Section 7, but no solution seems to provide the degree of control we have found necessary. Thus, we have adapted several mechanisms—dynamic regions, unique pointers, and reference-counted objects—each with its own strengths and weaknesses, to provide programmers a better set of tradeoffs. The following sections discuss these new mechanisms.

3. DYNAMIC REGIONS

Our first addition to Cyclone is a form of *dynamic regions* inspired by the work of Hawblitzel and von Eiken [25]. Dynamic regions, like lexical regions, are containers that allow allocation of individual objects, but only deallocation of the entire container. Unlike a lexical region, a dynamic region can be explicitly deallocated at (almost) any program point.

To ensure that a dynamic region is not accessed after it has been deallocated, we associate extra state with the region that must be checked at runtime before granting access to the region. If the region has been deallocated, the access check fails by throwing an exception. This check is analogous to a checked type cast to a live, lexical region.

To avoid checking the state each time a dynamic region is accessed, we provide a lexically scoped open construct, as this example demonstrates:

```
void foo(dynregion_t<‘r,‘H> k) {
  int *‘r x;
  { region h = open(k);  //grants access to ‘r
    x = rmalloc(h, sizeof(int));
    *x = 42;
    bar(h,x);
  }
  free_dynregion(k);  //destroys ‘r
}
```

The function takes a parameter k that is a *key* for the dynamic region ‘r. The key contains the state indicating whether the region has been deallocated, as well as a reference to the region itself. This state must persist beyond the lifetime of the region because it may be consulted after the region has been deallocated. In this example, the state is stored in the heap region (‘H), but, in general, the state can be allocated anywhere. The key can be used only when the region in which it resides is known to be live.

In the example above, ‘r is a region name in scope in foo. However, it is *not* assumed to be live upon entry to the function—by default, regions occurring within dynregion_t are not assumed live (though an explicit effect can indicate otherwise). Thus, any attempt to dereference a pointer into ‘r will be rejected by the type checker. The open construct allows access to a dynamic region given a key. In particular, region h = open(k); S takes a key k, checks that the region has not been deallocated, and if so, binds a handle for the region to h. Access to the region is granted throughout the scope of the statement S. Thus, in the scope of an open, one can freely allocate, dereference and pass to functions pointers into the region, exactly as though it were a lexical region.

The primitive `free_dynregion` takes a key and reclaims the storage of the associated region, updating the key's state to record that the region is no longer accessible. Thus, subsequent attempts to `open` will result in an exception. If the region is `open` or it has already been deallocated, then `free_dynregion` fails.

Adding dynamic regions to Cyclone was extremely simple, as we already had an effect system to keep track of regions that can be safely accessed. Indeed, we can think of the lexical-region declaration `region<‘r> h;` S as an abbreviation for creating a dynamic region, opening it for the scope of S, and calling `free_dynregion` upon exit from S. The user is never given access to the key for `‘r`, so S cannot deallocate the region, but it can be safely deallocated outside of S.

When coupled with existential types, dynamic regions are fully first class: they can be placed in data structures (e.g., a hash table) and deallocated at will (e.g., when removing an item from the table).

Dynamic regions have some drawbacks. First, unlike lexical regions, there is a potential for an exception to be thrown when opening or freeing a dynamic region. Second, the key state for a dynamic region (12 bytes in our current implementation) has to be stored somewhere and can become a source of leaks. For instance, we can code an iterative algorithm, such as Wang and Appel's copying collector, but we end up leaking a key for each collection. A unique pointer to the key can prevent this leak and is an important synergy of our mechanisms.

4. UNIQUE POINTERS

Lexical and dynamic regions are not efficient memory management mechanisms for small sets of objects, or for sets of objects that need to be deallocated at widely varying times. Cyclone's *unique pointers* address these situations by providing for the safe and efficient deallocation of individual objects using `free`.

In the presence of aliases, `free` can lead to unsafe programs. In particular, calling `free(x)` may deallocate an object referred to by another variable `y`, introducing a dangling pointer. By limiting use of `free` to unique—unaliased—pointers, we avoid the problem.

Like a dynamic region, the object that a unique pointer points to can be deallocated at will. Unlike a dynamic region, there is no run-time state to ensure that subsequent accesses are prevented. Instead, we rely on a conventional flow analysis to ensure that an object is never accessed once it has been deallocated. The analysis is greatly simplified by disallowing copies of unique pointers. More properly, at any program point, there is at most one (usable) copy of a value assigned a unique-pointer type. If that pointer's object is freed, then we need not worry about preventing access through an alias.

The idea of using unique pointers is derived from linear and affine type systems, and has been suggested in many other settings (see Section 7). However, we found that a conventional approach to linearity was far too restrictive. In particular, a conventional linear type system prohibits placing linear objects inside nonlinear objects. Furthermore, a conventional linear type system forces the user to follow awkward coding idioms. For instance, to calculate the length of a list, the list must be torn apart and reconstructed. Finally, the introduction of linearity complicates type abstraction (i.e., polymorphism) since we must distinguish linear and nonlinear types. In turn, it becomes difficult to write reusable libraries.

Our design extends conventional approaches to linearity in three key respects:

1. We allow unique pointers to be embedded within shared objects, and provide an atomic swap operator that lets them be accessed safely.

2. We provide support for temporarily treating a collection of unique pointers as if they were pointers into a lexical region. Hence we can reuse code for "reader" functions (e.g., calculating a list's length) without using awkward coding idioms.

3. We provide additional polymorphism to let us abstract over types that can contain unique pointers or nonunique pointers.

The following sections discuss these aspects of our design.

4.1 Simple Unique Pointers

A unique pointer can be created by calling `malloc` and destroyed by calling `free`. To distinguish unique pointers from pointers into a lexical or dynamic region, we use types of the form τ `*‘U`. Here, `‘U` is a distinguished region name that indicates uniqueness. Semantically, we think of τ `*‘U` as an abbreviation for (ν`‘r.` τ `*‘r`) where we interpret the binding ν`‘r` as meaning "there exists a fresh region `‘r`." In other words, each unique pointer is conceptually a reference into a region that contains a single object, and that region is distinct from any other region.

As a simple example, we can write:

```
struct point { int x; int y; } *‘U p;
p = malloc(sizeof(struct point));
p->x = 1;
p->y = 2;
...
free(p);
```

This code declares `p` to be a unique pointer to a point, allocates storage for the point, initializes its components, and ultimately frees it.

An intraprocedural, flow-sensitive, path-insensitive analysis guarantees that variables and components of data structures are *defined* before they are used. The analysis is a largely straightforward abstract interpretation that operates over a heap abstraction that includes must points-to information. (The details of the analysis are described in Grossman's dissertation [20].) The important point for this paper is that a unique pointer can become *consumed* (e.g., by passing it to `free`), in which case the analysis signals an error if there is a subsequent attempt to use it. We chose an intraprocedural analysis to ensure that type-checking remains modular, and a path-insensitive analysis to ensure scalability.

To simplify the analysis further, we ensure that there is at most one usable copy of a unique pointer value by treating copies as destructive. For instance, if `p` is a unique pointer variable, and we assign its value to `q`, then in the continuation, `p` is considered to be consumed. This ensures that if we call free on q, the deallocated object cannot be accessed through the alias p. At run-time, we do not actually destroy the reference in p. Reading through a unique pointer (e.g., *p or p->x) does not consume it.

By default, the analysis considers unique pointers passed to function calls as consumed, expecting the callee to deallocate the value, return it to the caller, or place it in a data structure. This treatment can be overridden with an explicit noconsume attribute on the function's prototype. If present, the caller is ensured that the value is still defined upon return, and the callee cannot consume the value.

At join points in the control-flow graph, our analysis conservatively considers a value consumed if there is an incoming path on which it is consumed. For instance, if p is a defined unique pointer and we write:

```
if (rand()) free(p);
```

then in the continuation, the analysis treats p as being consumed. Unfortunately, this can lead to leaks, so we issue a warning in this situation (and a few others such as overwriting a defined unique pointer). We could generate an error instead, but we have found that this results in too many type errors, primarily because of exception handlers. These handlers typically have a large number of incoming control-flow edges (at least one for each function call within the scope of the handler) and it is almost never the case that the same unique pointers have been consumed on every edge.

A few other details are necessary to ensure the system is sound. First, we must prevent pointer arithmetic or expressions like &p->y when p is a unique pointer because free expects a pointer to the beginning of the object. Second, polymorphism must be treated with some care, as we discuss in Section 4.3.

Finally, we must ensure that copies of unique pointers are made only along *unique paths*. A unique path u has the form

u ::= x | u.m | u->m | *u

where x is a local variable, and u is a unique pointer. To appreciate the unique-path restriction, consider this incorrect code:

```
int f(int *‘U *‘r x) {
  int *‘U *‘r y = x;  //x and y are aliases
  int *‘U z = *y;
  free(z);
  return **x;  //accesses deallocated storage!
}
```

Here, x is a pointer into a conventional region ‘r and thus its value can be freely copied into y. We then extract a unique pointer from the contents of y and free it. Then we attempt to access the deallocated storage through x.

In most languages based on linear types, this problem is avoided by requiring that linear objects cannot be placed in nonlinear containers. Our approach is similar, except that we forbid copying of a unique value unless the path to the value is unique. In the example above, the attempt to initialize z with *y is a compile-time error.

4.2 Unique Pointers in Shared Data

With no additional access mechanism, the unique-path restriction prevents using a unique pointer that is placed within a shared object, which is too restrictive. For instance, we could never use a unique pointer stored in a global variable. To overcome this limitation, we provide an atomic *swap* operation, written e_1 :=: e_2. The addition of swap was inspired by Baker's work on a linear variant of LISP [6]. In Cyclone, swap can be performed on any pair of (left-hand-side) expressions of unique-pointer type, including paths that go through nonunique pointers. It is roughly equivalent to, "temp = e_1; e_1 = e_2; e_2 = temp;" The intuition behind the soundness of swap is that it preserves our crucial invariant: at any program point there is at most one usable copy of a unique pointer value. This idea is formalized in our work on linearly typed assembly language [14] and can also be justified with formalisms such as alias types [37].

Here is a simple example of the utility of swap:

```
int *‘U g = NULL;
void init(int x) {
  int *‘U temp = malloc(sizeof(int));
  *temp = x;
  g :=: temp;
  if (temp != NULL) free(temp);
}
```

Here, g is a global variable that holds a unique pointer to an int. The init routine creates the unique pointer and stores it in a temporary variable. Then, the value of the temporary is swapped for the value of g. After the swap, if temp is not NULL, then we free the pointer. It is easy to verify that at any program point, there is at most one usable copy of any unique value. Furthermore, since the swap is atomic, this property holds even if multiple threads were to execute init concurrently.

Our atomic swap operator makes it possible to build a set of protocols for shared, concurrent objects without losing the advantages of local reasoning afforded by unique pointers. An obvious extension is to provide a form of compare-and-swap so that we could build arbitrary wait-free structures [28].

4.3 Polymorphism

Cyclone supports polymorphism, which is crucial for writing reusable library functions. With some care, Cyclone's polymorphism can be extended to handle unique pointers. The following function illustrates some of the difficulties. It takes a (nonempty) list and turns it into a circular list:

```
typedef struct List<‘a,‘r> *‘r list_t<‘a,‘r>;

list_t<‘a,‘r> cycle(list_t<‘a,‘r> x) {
  list_t<‘a,‘r> res = x;
  while (x->tl != NULL) x = x->tl;
  x->tl = res;
  return res;
}
```

The full type of the function might informally be written

∀‘a::BT,‘r::R.list_t<‘a,‘r> → list_t<‘a,‘r>

where ‘a ranges over boxed types, indicated by the kind BT, and ‘r ranges over regions, indicated by the kind R.

Circular lists clearly violate our uniqueness invariant, so, we do not expect cycle to work on lists allocated in ‘U. Indeed, if we instantiate ‘r with ‘U, the body of the function does not typecheck, because x becomes consumed at the assignment to res, so it cannot be used in the while loop. To prevent this, we make a distinction between ‘U and other regions: we make R the kind of nonunique regions, and we have a separate, incompatible kind UR for ‘U.

A different problem arises if we attempt to instantiate a type variable with a unique-pointer type. Consider:

```
‘a hd(list_t<‘a,‘r> x) { return x->hd; }
```

If we instantiate ‘a with int *‘U, the code does not type check because we access a unique pointer via a nonunique path. To avoid this problem, we introduce a kind distinction between unique pointer types (UBT) and other boxed types.

These distinctions are sufficient to make our polymorphism safe, but they do not help us as much as we would like. For example, the length function of Section 2 applies only to lists of elements that are not unique pointers. We can write a version for lists of unique pointers just by changing the kind of the element type to UBT, but that version would not work on lists with nonunique elements.

To address this, we further augment the kind system by adding "top" elements to type and region kinds. The kind TopR ranges over unique and non-unique regions, and the kind TopBT ranges over all boxed types, resulting in a natural sub-kinding lattice for both regions and types:

TopR
R
UR
TopBT
BT
UBT

The top kinds are restricted by all of the constraints imposed by their subkinds. For instance, a value of type ‘a::TopBT cannot be freely duplicated, must be accessed via unique paths or a swap, and cannot be freed. We note that kinds and sub-kinding were already necessary in Cyclone to distinguish types from regions, and boxed types from other types. Fortunately, default kinds and kind inference minimize the programmer's burden.

Top kinds make it possible to write functions that are polymorphic over uniqueness. For instance, the following function destructively reverses lists:

```
list_t<‘a::TopBT,‘r::TopR>
  imp_rev(list_t<‘a,‘r> x) {
    if (x == NULL) return NULL;
    list_t<‘a,‘r> y = NULL;
    x->tl :=: y;
    while (y != NULL) {
      list_t<‘a,‘r> temp = NULL;
      temp :=: y->tl;
      y->tl = x;
      x = y;
      y = temp;
    }
    return x;
  }
```

Careful examination shows that the code is well-typed, regardless of the boxed type we use to instantiate ‘a or the kind of region we use for ‘r.

Unfortunately, the restrictions imposed by the top kinds prevent us from writing many useful polymorphic functions. For example, many functions need to alias their arguments internally, in a way that is not visible to the caller. It should be safe to call such a function with a unique pointer, but this will not be permitted by the kind discipline we have described. The next section gives a solution to this problem.

4.4 Temporary Aliasing

Programmers often write code that aliases values temporarily, e.g., by storing pointers in loop iterator variables or by passing them to functions. Even with noconsume, such reasonable uses would be severely hampered by the system presented thus far. To address this problem, we introduce a primitive called alias that permits temporary aliasing of a unique pointer for the duration of a statement block, provided that no aliases are live when the block completes. This primitive resembles and extends Walker and Watkins' let! [43], the unpack primitive of alias types [37], and Clarke's notion of borrowing [15]. Here is a contrived example:

```
void inc(int *‘r1 cell) {
  int *‘r1 t = cell;
  print_cell(t);
  *cell = *t + 1;
}

void g() {
  int *‘U xptr = malloc(sizeof(int));
  *xptr = 3;
   { alias <‘r2> int *‘r2 temp = xptr;
     inc(temp);
   }
   free(xptr);
}
```

Imagine that inc is an existing, widely-used library function that was not written with the constraints of uniqueness in mind. In this simple example, it copies its pointer argument (using both copies) and passes its pointer argument off to another function (print_cell). Thus, inc would not be well-typed if we replaced ‘r1 with ‘U, so ‘r1 is restricted to nonunique regions (kind R).

The function g creates a unique pointer xptr that it wishes to pass to inc. It does so by using an alias declaration to (a) introduce a fresh region variable ‘r2 of kind R and (b) introduce an alias for xptr in the locally-bound variable temp. The temp alias is assigned the type int *‘r2 and can thus be passed to inc and freely copied. The original unique pointer, xptr, is considered consumed for the duration of the block. Thus, it is impossible for the value to be freed during the execution of the declaration's block. At the end of the alias block, any copies of the unique pointer become unusable, since ‘r2 goes out of scope. This allows us to once again treat xptr as if it is a unique pointer so that we can, for instance, pass it to free.

In short, alias lets us temporarily treat a unique pointer as if it were a pointer into a conventional region, without losing the ability to recycle the storage later. Throughout the scope of the alias, we can make copies of a pointer, place it in conventional (shared) data structures, etc. The fresh region name, ‘r2, ensures that no (usable) copies escape the scope of the construct.

Viewed from another perspective, the flow analysis and type system are preventing the unique pointer from being deallocated, at least temporarily. Thus, if we introduce a lexically scoped region ‘r2, the unique pointer will always *outlive* ‘r2. Thus, it is safe to treat τ *‘U as a subtype of τ *‘r2. Indeed, it is sound to extend this subtyping relation *through read-only type constructors*, so that we can treat an indeterminate number of unique pointers as if they were references into ‘r2.

For example, consider the following definitions:

```
struct CList<‘a,‘r> { // read-only lists
  ‘a hd;
   struct CList<‘a,‘r> *const ‘r tl;
};
typedef
  struct CList<‘a,‘r>*const ‘r clist_t<‘a,‘r>;
int clength(clist_t<‘a,‘r::R> x);
int ulength(list_t<‘a,‘U> x) __noconsume(1)__ {
  int res;
  { alias<‘r2> clist_t<‘a,‘r2> t =
        (clist_t<‘a,‘U>)x;
    res = clength(t);
  }
  return res;
}
```

The clist_t constructor is the same as list_t except that the list spine must be read-only. The clength function takes a read-only list where each cons cell is, as far as the function is concerned, allocated in a nonunique region ‘r. Through conventional subtyping, it is possible to pass a list_t, allocated in some nonunique region to clength. That is, list_t<‘a,‘r> is a subtype of clist_t<‘a,‘r> for any type ‘a and any region ‘r.

However, it is also possible to pass a unique list_t to clength as shown by the function ulength. In that function, we first coerce the value x to a read-only list. We then bind it with the alias construct to a temporary that allows us to promote the clist_t<‘a,‘U> value to a clist_t<‘a,‘r2> value. We then pass the ‘r2 version to clength. At the end of the function, we are ensured that x is not consumed which is required due to the noconsume attribute. In turn, this ensures that the caller can continue to use, and ultimately free, the list.

What is the intuition behind the soundness of such a "deep" alias? It is clear that region scoping prevents any copies of the pointers from escaping. By assigning these tail pointers nonunique pointer types, we are preventing some function from deallocating one of the cells throughout the call to length. Furthermore, because we have a unique root for the data structure (i.e., exclusive ownership), there can be no other way to get to these values and free them.

It may seem that the read-only requirement is too strong, but the counterexample below shows its necessity. In the example, we overwrite one of the unique pointers with another to create a circular list by taking advantage of alias. The type-checker would not reject the assignment since we have temporarily given all the unique pointers the same type (a list pointer into region ‘r). But on exit from the alias, we free what the tail of the list points to, namely the list itself. We then attempt to access the deallocated storage. To prevent this problem, we must therefore restrict deep aliasing to read-only paths. This is not surprising as deep subtyping, in general, is restricted in the same fashion.

```
‘a foo(list_t<‘a,‘U> x) {
  { alias<‘r> list_t<‘a,‘r> temp = x;
    temp->tl = temp; //unsound: creates cycle!
  }
  list_t<‘a,‘U> tail = x->tl;
  free(tail);
  return x->hd;
}
```

For improved programmer convenience, the Cyclone type-checker optimistically inserts alias blocks around function-call arguments that are unique pointers when the formal-parameter type is polymorphic in the pointer's region. If this modified call does not type-check, we remove the inserted alias. For example, one can rewrite the ulength function from the previous section as simply:

```
int ulength(list_t<‘a,‘U> x) __noconsume(1)__ {
  return clength(x);
}
```

This backtracking scheme is much like Aiken et al.'s approach for inferring uses of a similar confine construct [3].

We have not yet proven the soundness of our alias construct, though we are confident that it is true. As mentioned previously, the shallow version of alias can be seen as a version of Walker and Watkins' let!. However, we have left the soundness of deep alias to future work.

5. REFERENCE-COUNTED OBJECTS

Reference counting is often used to track the lifetimes of shared objects in systems applications; for example, it is used in both COM and in the Linux kernel. Cyclone supports a form of reference counting that builds on unique pointers. This has two great advantages: First, we introduce almost no new language features, rather only some simple run-time support. Second, the hard work that went into ensuring that unique pointers coexisted with conventional regions is automatically inherited for reference-counted objects.

We define a new *reference-counted region* ‘RC, whose objects, when allocated, are prepended with a hidden reference-count field. As with unique pointers, the flow analysis prevents the user from making implicit aliases. Instead, ‘RC pointers must be copied *explicitly* by calling alias_refptr, which has type:

```
‘a *‘RC alias_refptr(‘a *‘RC) __noconsume(1)__;
```

Calling alias_refptr creates an alias and increases the reference count of the underlying object. The noconsume attribute specifies that the caller can still use the original pointer, as well as the newly returned pointer. In essence, they are both explicit capabilities for the same object.

A reference-counted pointer is destroyed by a call to:

```
void drop_refptr(‘a *‘RC);
```

This consumes the given pointer and decrements the object's reference count; if the count becomes zero, the memory is freed. As with unique pointers, the flow analysis warns when an ‘RC pointer is potentially "lost" at a control-flow join point. This helps ensure that we do not forget to decrement the counter on some path. Most importantly, we guarantee a pointer is not prematurely freed due to a mismanaged count.

We assign ‘RC the kind TopR. Thus pointers into it are treated the same as unique pointers, except they cannot form part of a unique path, and cannot be passed to free. Thus, a function such as imp_rev (Section 4.3) that abstracts over TopR can be passed a reference-counted object.

6. PROGRAMMING EXPERIENCE

Cyclone has been used for several projects where safety is important and designers felt garbage collection was inappropriate [34, 33, 10]. We have used the language to build the Cyclone compiler, and a large collection of libraries and tools. In this section, we describe our overall assessment of Cyclone's memory-management support, followed by more detailed experiences with an event-based web server and in an overlay network for streaming data [29]. We also present performance results demonstrating the ability to control memory consumption from within our language.

6.1 Overall Experience

Not surprisingly, code that does only heap allocation and relies upon garbage collection is the easiest to write and maintain. On the other hand, we generally found that we could improve performance and/or space overheads by judicious application of the other options.

Stack and lexical region allocation are relatively easy to use, due to the local region inference, the carefully chosen default effects, and the fact that we developed most of our libraries with region allocation in mind. For instance, the string, standard I/O, list, and hashtable libraries all expect region-allocated data. There are annoying aspects, such as having to parameterize type definitions by a suitable number of regions, and having to pass region handles to the right functions. Support for nested functions (i.e., closures) would ease the latter considerably.

Dynamic regions are as easy to use as lexical regions, and sometimes easier. For instance, dynamic region keys can be placed in global variables that hold cached results, such as lexemes in our compiler.

Our initial design for unique pointers had no support for `alias` or placing unique pointers in shared objects. We quickly found this design unusable. When we added support for these features, coding become easier, though still somewhat tedious. With the addition of our primitive `alias` inference, writing code became *much* easier.

Nonetheless, room for improvement remains. For instance, our `alias` inference is restricted to function call contexts. In MediaNet, inference discovers 71% of the 66 needed `alias` statements. Of the ones that remain, the majority are due to the need to perform pointer arithmetic on or take the address of unique pointers. A more general constraint-based inference could discover these and other uses. Similarly, support for a `restrict` mechanism in the style of Aiken et al. [3] might help eliminate the need for swapping, at least for single-threaded code.

6.2 Web Server

We built a simple, space-conscious web server to demonstrate how unique pointers give Cyclone programmers fine-grained control over memory use. The web server allocates its objects either statically, on the stack, or with unique pointers. Consequently, it does not need a garbage collector at all, and we linked it with the Lea allocator [30] instead.

The server is single threaded, and supports concurrent connections using non-blocking I/O and an event library in the style of libasync [31] and libevent [35]. After opening a socket to listen for HTTP connections, the server enters an event loop that dispatches ready file descriptors or signals to callbacks registered by the server. A callback is implemented as a closure consisting of a pointer to a function and an environment that is passed to the function when it is called. Because concurrent HTTP connections overlap in a non-nested fashion, we used unique pointers to implement closures and environments, rather than using our lexical region constructs.

Our callbacks are implemented with Cyclone structs:

```
struct CB {  <‘a::TopBT> : regions(‘a) > ‘H
  void (*f)(int,short,‘a);
  ‘a env;
};
```

Here, `‘a` is an existentially-bound type variable that represents the type of the environment, and `f` is a function pointer that expects the environment `env` of type `‘a` to be passed as its third argument. The first argument of `f` will be the ready file descriptor or signal, and the second argument tells the function whether the first argument is a descriptor ready for reading, a descriptor ready for writing, or a signal. The environment type `‘a` is declared with kind `TopBT`, which is the kind of boxed types that are potentially unique pointers. (The outlives constraint "`regions(‘a) > ‘H`" is necessary in practice as described in our previous work [22], but for simplicity, we ignore it here.) In our web server, environments are either integers or unique pointers to compound objects. When the environment is a unique pointer, our convention is that the callback itself is responsible for freeing the environment if necessary.

File-descriptor callbacks are registered with the `fdcb` function, which has the following type:

```
void fdcb(int fd, short ev, struct CB *‘U cb);
```

For example, `fdcb(5,EV_READ,cb)` registers a callback that waits for input (indicated by the constant `EV_READ`) on file descriptor 5. Here `cb` is a unique pointer to a callback structure that the caller must allocate. The callback structure is freed by the event loop when the callback is invoked.

Our web server is optimized for space usage. When a file is requested by a client, the server allocates a small buffer and uses the buffer to read the file and send it to the client in chunks. We used a 1KB buffer size for our measurements, but of course the size is configurable. Figure 1 demonstrates the tight control over memory that we achieved, by tracking the memory use of the web server under a sustained load with a maximum of 40 concurrent connections. The x-axis plots CPU time in terms of clock ticks (as determined by the `clock()` system call), while the y-axis plots memory consumed. We also plot the total space reserved by the allocator (i.e., acquired from the operating system). Our profiler confirms that all dynamic memory is stored in the unique region, which occupies at most 40KB or so (1KB per 40 connections) of the total reserved memory of 44KB. The server thus makes very efficient use of heap memory, with little fragmentation. And, of course, there are no pauses introduced by garbage collection.

6.3 MediaNet

MediaNet is an overlay network for performing on-line, adaptive scheduling for packet streams with user-specified resource constraints [29]. Each node in the network runs a local server, implemented in Cyclone, that communicates with the other servers to deliver and adaptively transform streaming data. Each local server behaves according to a configuration program called a *Continuous Media Network* (CMN). This is simply a directed acyclic graph (DAG) of *operations*, where each operation works on the data as it passes through. As network conditions change, a global scheduler may reconfigure local schedulers to implement better-performing CMNs. On each local scheduler, the new CMN

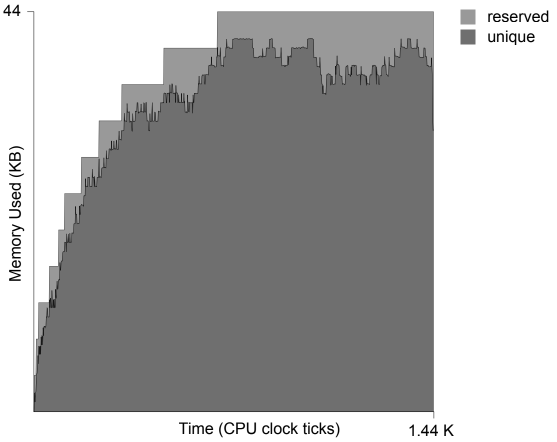

Figure 1: Memory use of the web server with up to 40 concurrent clients

will begin to run alongside the old one, until all old data has been delivered and the old CMN can be removed.

In the local-scheduler implementation, we allocate CMNs in dynamic regions; the currently-active CMN is in the *current* region, while the new CMN, present only during reconfiguration, is in the *new* region. After reconfiguration, the *current* region can be freed, and the *new* region becomes *current*. Regions work well for CMNs because all the relevant data is allocated and logically deleted at the same time. Dynamic regions are necessary because the lifetimes of the current and new CMNs overlap, but are not nested.

The packets sent between operations are implemented as a simpler variant of Linux's `skbuff`s, called *streambuffs*:

```
struct StreamBuff { <i::I>
  ... // three omitted header fields
  tag_t<i> numbufs;
  struct DataBuff<‘RC> bufs[numbufs];
};
```

The packet data is stored in the array `bufs`. Note that `bufs` is not a pointer to an array, but is flattened directly within `StreamBuff`. Thus `StreamBuff` elements will vary in size, depending on the number of buffers in the array. The `numbufs` field holds the length of `bufs`. The notation `<`*i*`::I>` introduces an existential type variable that has integer kind `I`, and is used by our type system to enforce the correspondence between the `numbufs` field and the length of the `bufs` array. *Databuffs* store packet data:

```
struct DataBuff<‘r> {
  unsigned int ofs;
  char ?‘r buf;
};
```

The `buf` field points to an array of the actual data. The `?` notation designates a pointer to a dynamically-sized buffer, which is accompanied by bounds information to prevent overflow. The `ofs` field indicates an offset, in bytes, into the `buf` array. This offset is necessary when `‘r` is `‘U` or `‘RC` since pointer arithmetic is disallowed in those cases; the `StreamBuff` definition allocates `buf` in `‘RC`.

While databuffs are reference-counted, we allocate streambuffs uniquely, so they can be freed immediately after the corresponding data is sent. When multiple streambuffs must refer to portions of the same packet data, we clone them as shown in Figure 2. Here, three individual streambuffs A, B, and C share some underlying data; unique pointers have open arrowheads, while reference-counted ones are filled in. This situation could have arisen by (1) receiving a packet and storing its contents in A; (2) creating a new buffer B that prepends a sequence number 1234 to the data of A; and (3) stripping off the sequence number for later processing (assuming the sequence number's length is 4 bytes). Thus, C and A are equivalent. When we free a streambuff, we decrement the reference counts on its databuffs, so they will be freed as soon as possible.

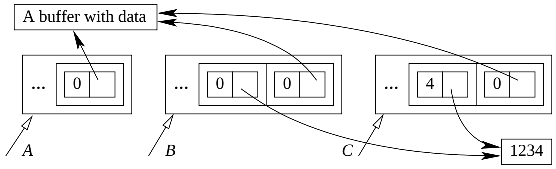

Figure 2: Pointer graph for three streambuffs

An earlier version of MediaNet stored all packet data in the garbage-collected heap, and used essentially the same structures for packet data. One important difference was that databuffs contained an explicit `refcnt` field managed by the application to track aliasing. If an operation determined that no aliases to a packet's data existed, the data could be safely mutated, improving performance. Unfortunately, this approach yielded a number of hard-to-find bugs whose appearance depended on configuration, data format, and timing. The current version uses `‘RC` pointers instead of manual counts. This greatly reduces the possibility of mismanaging the count, and lets us free the data immediately after its last use.

6.3.1 Performance

Although moving streambuffs and databuffs to unique pointers and reference counting does not eliminate MediaNet's reliance on the garbage collector, it does significantly improve performance. In a simple experiment, we used the TTCP microbenchmark [32] to measure MediaNet's packet-forwarding throughput and memory use for varying packet sizes. We measured two configurations:

- *gc+free* is MediaNet built as described above, using the Boehm-Demers-Weiser (BDW) conservative garbage collector [9], version 6.2α4, for garbage collection and manual deallocation.

- *gc* is as above, but with streambuffs and databuffs stored in the garbage-collected heap.

For our experimental setup, we used three 1 GHz Pentium III's, each running Linux kernel 2.4.18 with 250 MB of RAM. The machines were directly connected in a line via gigabit Ethernet (using Intel Pro/1000 F cards), with the middle machine acting as a router. The MediaNet server ran on this machine, and the TTCP sender and receiver ran on opposite ends.

Figure 3 plots the total throughput of MediaNet, in megabits per second, as a function of packet size (note the logarithmic scale). Each point is the median of 21 trials in which 5000 packets are transferred, with little variance: the

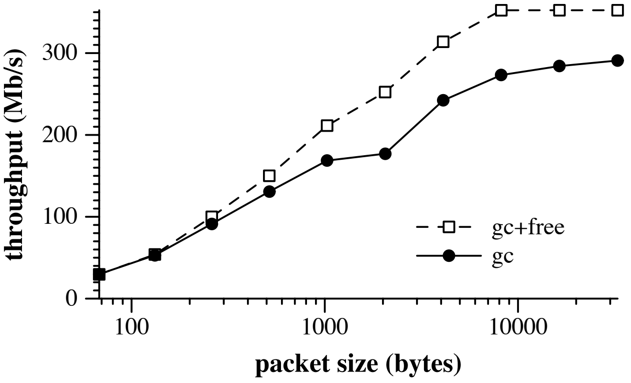

Figure 3: MediaNet throughput

semi-interquartile range[2] is typically less than 0.1% of the median. The two configurations perform roughly the same for smaller packet sizes, but *gc* starts to fall behind as packets become larger than 512 bytes. The largest gap is for 2 KB packets, where the *gc+free* case achieves 42% better throughput; at 32 KB packets the improvement is 21%.

Figure 4 illustrates the memory usage of each configuration for the experiment in which 5000 4 KB packets are transferred. This graph has the same format as the graph in Figure 1, but additionally shows the heap and reference-counted regions. Also, the reserved memory for the *gc+free* case is not shown.

The *gc* configuration exhibits a sawtooth pattern, where each peak roughly coincides with a garbage collection. Interestingly, the locations of the peaks also exhibit a sawtooth trend; the BDW collector often collects before all available memory is exhausted, and delays some work to reduce pause times. The large gap between the topmost peak and the amount of reserved data is evidently fragmentation. The *gc+free* configuration both uses and reserves far less memory (128 as opposed to 840 KB for reserved memory, and 8 as opposed to 420 KB of peak used memory) There is some initial data allocated in the heap that stays constant through the run, and the reference-counted and unique data (the small line at the bottom) never consume more than a single packet's worth of space, since each packet is freed before the next packet is read in.

For comparison, we also ran our experiment using the Lea allocator. It performed slightly better than *gc+free*, exhibiting slightly higher throughput, and reserving less memory (only 25 KB as opposed to 128 KB).

7. RELATED WORK

The ML Kit [38] implements Standard ML with regions. Whole-program analysis (type inference) assigns regions using a system that (like Cyclone) has LIFO regions as its backbone [39]. Extensions to avoid some LIFO restrictions include an analysis for late allocation and early deallocation of regions [2], integration with an accurate garbage collector [23], and a region `reset` analysis.

The RC language and compiler [19] provides language support for regions in C. Access control for regions is accomplished by dynamic reference counts instead of static type tests, though an analysis tends to eliminate much of the overhead. RC does not prevent dangling pointers to data not in regions, so there is no support for ensuring conventional uses of `malloc`/`free` are safe.

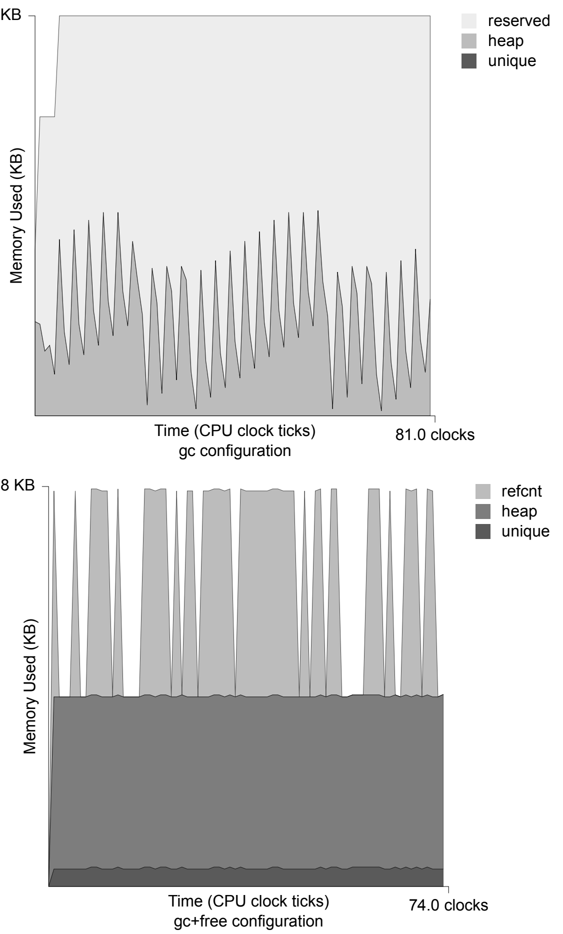

Figure 4: MediaNet memory profile (4 KB packets)

Work by Bacon et al. [5] and Boyapati et al. [11] to prevent data races in Java uses unique pointers as one way to prevent two threads from simultaneously accessing the same object. These projects use special syntax for "destructive reads" (swapping in NULL). Boyapati et al. [12] have recently used a region-based type system for avoiding run-time errors in Real-Time Java [7] programs. Real-Time Java regions are like Cyclone's dynamic regions but have a more awkward semantics. A region is implicitly deallocated when no thread has it opened and a rather *ad hoc* "portal" field is NULL. Without portals, threads would have no way to share memory that did not outlive one of the threads. With unique pointers to dynamic-region keys, Cyclone programmers can encode portals.

Work on linear types [40], alias types [37, 42], capabilities [41], and linear regions [43, 27] provide important foundations for safe manual memory management on which we have built. In making these ideas convenient in a source language, we have needed interesting extensions like `alias` and reading through unique pointers without consuming them.

[2] The semi-interquartile range is similar to the standard deviation, but is relevant when choosing the median as the single-point summarizer.

Vault [16, 17] is another project adapting work on regions and linearity to a source language. Unique pointers allow Vault to track sophisticated type states, including whether memory has been deallocated. To relax the uniqueness invariant, they use novel *adoption* and *focus* operators. Adoption lets programs violate uniqueness by choosing a unique object to own a no-longer-unique object. Deallocating the unique object deallocates both objects. Compared to Cyclone's support for unique pointers in nonunique context, adoption prevents more space leaks, but the semantics requires hidden data fields so the run-time system can deallocate large data structures implicitly. Focus allows adopted objects to be temporarily unique. Compared to *swap*, focus does not incur run-time overhead, but the type system to prevent access through an unknown alias requires more user annotations. That said, the type system appears expressive enough to encode swap. Compared to *alias*, focus is less powerful because it applies only to a single object. Focus also does not work as-is with multithreading, whereas implementing swap atomically makes our approach sound in a multithreaded setting. Integration with Cyclone's multithreading design [21] remains future work.

Numerous projects have enriched imperative languages with unique pointers using destructive reads to preserve uniqueness. Using swaps instead of an implicit NULL is rare, but has been done [6, 24]. Most systems allow temporary aliasing of an individual object, but nothing like our "deep" `alias`. Clarke's recent work on external uniqueness [15] uses *ownership types* to ensure references do not escape the scope of a temporary alias. The technique is similar to our use of regions, but the lack of an effects system leads to different design decisions. Boyland [13] summarizes several projects and proposes using static analysis to avoid the disadvantages of destructive reads. An intraprocedural analysis can allow aliases of unique objects so long as multiple aliases are not used on any program path.

Uniqueness types in the functional language Clean [1] allow in-place update and functional I/O. Such types can refer only to values pointing to objects not otherwise referenced. A flow-sensitive "sharing" analysis ensures this restriction.

Berger et al.'s *reaps* [8] combine the run-time performance advantages of regions (batched deallocation) with individual objects (fine-grained deallocation). They permit deallocating objects within regions and report performance superior to application-specific allocators. Reaps validate the importance of regions and individual objects, but they do not prevent dangling-pointer dereferences.

Finally, sophisticated interprocedural analyses are starting to appear to detect leaks (e.g., [26]) or more generally reason about temporal heap properties (e.g., [36]). It is not yet clear if they are cheap enough to run on every compilation or if they can give the strong safety guarantees of Cyclone's intraprocedural analysis, especially in the presence of threads and/or separately compiled libraries. On the other hand, these analyses typically need far fewer annotations.

8. CONCLUSIONS

Cyclone now supports a rich set of safe memory-management idioms for users unwilling to use only automatic techniques:

- *Stack/lexical regions:* We can avoid any run-time cost for data whose lifetime is known sufficiently well when allocated.
- *Dynamic regions:* We can aggregate the run-time cost and potential failures for data that can be deallocated simultaneously.
- *Heap region:* We can use conservative garbage collection for a portion of a program's data.
- *Uniqueness:* We can support manual deallocation of unaliased data. We can put unique pointers in nonunique data structures by using a swap operator to access them.
- *Reference counting:* We can support explicit copies of otherwise unaliased data and reclaim the data when no copies remain.

Users can use the best idioms for their application.

Moreover, we have designed linguistic constructs for tying these idioms together in a coherent language that supports reusable code amid well-known tradeoffs. Lexical regions are the backbone of our system and exploit the convenience of data lifetime corresponding to scope. We regain this convenience for dynamic regions with `open` and for unique and reference-counted pointers with `alias`. The latter extends previous approaches by allowing temporary aliasing of entire data structures. We also use polymorphism to write reusable code without temporary aliasing, but the coding style is often too awkward. Finally, we provide run-time checking when static enforcement is too onerous: Dynamic regions provide checkable keys to relax the compile-time constraints of lexical regions. Analogously, reference-counting provides checkable counts to relax the uniqueness invariant.

Together, these idioms represent significant progress toward our goal of enforcing sound, user-specified idioms. Looking forward, we envision a need for more specific aliasing information and more first-class status for reference counts. Nonetheless, we have been pleased with our ability to support natural invariants that improve actual application performance and predictability.

9. REFERENCES

[1] P. Achten and R. Plasmeijer. The ins and outs of Clean I/O. *Journal of Functional Programming*, 5(1):81–110, 1995.

[2] A. Aiken, M. Fähndrich, and R. 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.

[3] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *ACM Conference on Programming Language Design and Implementation*, pages 129–140, San Diego, CA, June 2003.

[4] Apache Foundation. Apache web server. `http://www.apache.org`.

[5] 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.

[6] H. Baker. Lively linear LISP—look ma, no garbage. *ACM SIGPLAN Notices*, 27(8):89–98, 1992.

[7] G. Bellella, editor. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[8] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–12, Seattle, WA, Nov. 2002.

[9] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software – Practice and Experience*, 18(9):807–820, 1988.

[10] H. Bos and B. Samwel. Safe kernel programming in the OKE. In *5th IEEE Conference on Open Architectures and Network Programming*, pages 141–152, New York, NY, June 2002.

[11] 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.

[12] C. Boyapati, A. Sălcianu, W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *ACM Conference on Programming Language Design and Implementation*, pages 324–337, San Diego, CA, June 2003.

[13] J. Boyland. Alias burying: Unique variables without destructive reads. *Software Practice and Experience*, 31(6):533–553, May 2001.

[14] J. Cheney and G. Morrisett. A linearly typed assembly language. Technical Report 2003-1900, Department of Computer Science, Cornell University, 2003.

[15] D. Clarke and T. Wrigstad. External uniqueness. In *International Workshop on Foundations of Object-Oriented Languages*, New Orleans, LA, Jan. 2003.

[16] 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.

[17] 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.

[18] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.

[19] D. Gay and A. Aiken. Language support for regions. In *ACM Conference on Programming Language Design and Implementation*, pages 70–80, Snowbird, UT, June 2001.

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

[21] D. Grossman. Type-safe multithreading in Cyclone. In *ACM International Workshop on Types in Language Design and Implmentation*, pages 13–25, New Orleans, LA, Jan. 2003.

[22] 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.

[23] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *ACM Conference on Programming Language Design and Implementation*, pages 141–152, Berlin, Germany, June 2002.

[24] D. Harms and B. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5):424–435, May 1991.

[25] C. Hawblitzel and T. von Eiken. Type system support for dynamic revokation. May 1999.

[26] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *ACM Conference on Programming Language Design and Implementation*, pages 168–181, San Diego, CA, June 2003.

[27] F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Principles and Practice of Declarative Programming*, Florence, Italy, Sept. 2001.

[28] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, Jan. 1991.

[29] M. Hicks, A. Nagajaran, and R. van Renesse. MediaNet: User-defined adaptive scheduling for streaming data. In *6th IEEE Conference on Open Architectures and Network Programming*, pages 87–96, San Francisco, CA, Apr. 2003.

[30] D. Lea. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[31] D. Mazières. A toolkit for user-level file systems. In *USENIX Annual Technical Conference*, pages 261–274, Monterey, CA, June 2001.

[32] M. Muuss. The story of TTCP. `http://ftp.arl.mil/~mike/ttcp.html`.

[33] P. Patel and J. Lepreau. Hybrid resource control of active extensions. In *6th IEEE Conference on Open Architectures and Network Programming*, pages 23–31, San Francisco, CA, Apr. 2003.

[34] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, and T. Stack. Upgrading transport protocols using untrusted mobile code. In *19th ACM Symposium on Operating System Principles*, Oct. 2003. To appear.

[35] N. Provos. libevent — an event notification library. `http://www.monkey.org/~provos/libevent/`.

[36] R. Shaham, E. Yahav, E. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with application to compile-time memory management. In *Static Analysis Symposium*, pages 483–503, San Diego, CA, June 2003.

[37] F. Smith, D. Walker, and G. Morrisett. Alias types. In *9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, Berlin, Germany, Mar. 2000. Springer-Verlag.

[38] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, and P. Sestoft. Programming with regions in the ML Kit (for version 4). Technical report, IT University of Copenhagen, Sept. 2001.

[39] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, Feb. 1997.

[40] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, Apr. 1990. North Holland. IFIP TC 2 Working Conference.

[41] 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.

[42] D. Walker and G. 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, Sept. 2000. Springer-Verlag.

[43] D. Walker and K. Watkins. On regions and linear types. In *6th ACM International Conference on Functional Programming*, pages 181–192, Florence, Italy, Sept. 2001.

[44] D. Wang and A. Appel. Type-preserving garbage collectors. In *28th ACM Symposium on Principles of Programming Languages*, pages 166–178, London, England, Jan. 2001.