Unique Pointers and Reference Counting in Cyclone

Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim

Abstract. Cyclone is a type-safe language intended for applications requiring control over memory management. Initial work on Cyclone supported stack allocation, lexical region allocation, and a garbage-collected heap. In this paper, we describe added support for unique pointers (which permit limited use of `malloc` and `free`) and reference-counted objects. These idioms are crucial for supporting overlapping, non-nested object lifetimes and safe manual deallocation. A region-based type-and-effects system ensures that dangling pointers cannot be dereferenced. Novel linguistic mechanisms leverage the region system to make it easier to use unique pointers. In particular, region polymorphism and a scoped construct for temporarily aliasing unique pointers let us write programs that can manipulate either unique or nonunique pointers. In our experience, our new constructs can significantly improve application performance, while adding a modest programming overhead.

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 memory management. Most safe languages allow memory to be recycled only through garbage collection. In Cyclone, garbage collection is not necessary. Rather, objects may be allocated on the stack or more generally in *regions* and reclaimed when the stack frame or region is deallocated. A type-and-effects system based on the work of Tofte and Talpin [30] ensures that pointers into deallocated regions are never dereferenced. However, region-based mechanisms have a serious weakness: they force object lifetimes to be nested. For example, stack-allocated objects and regions must obey a last-in-first-out discipline: only the most recently allocated object or region can be deallocated. Objects with overlapping, non-nested (ONN) lifetimes can be accommodated, but only at the cost of either allocating some objects earlier than otherwise necessary, or deallocating some objects later than necessary. ONN lifetimes are common in systems programs, like operating system kernels and event-based servers.

We have integrated two features into Cyclone for handling ONN lifetimes efficiently: reference counted objects, and a restricted (safe) version of `free`. These features allow us to safely deallocate an object at any time, without constraining the lifetimes of other objects. The key mechanism underlying both features is the *unique pointer*. By definition, a unique pointer is the only reference to an object, so freeing the object cannot make another reference a dangling pointer.

Thus, unique pointers give us a safe form of `free`. With a slightly different interpretation, they also give us safe, but programmer-controlled reference counting: incrementing the reference count of an object is like obtaining a unique pointer to a copy of the object, and decrementing the count is like freeing the copy.

To ensure safety, we must impose some restrictions on how unique pointers and pointers to reference counted objects can be accessed when placed in a shared (nonunique) object. We changed Cyclone's system of region polymorphism to reflect these differences, adding kinds that distinguish between unique pointers, nonunique pointers, and possibly-unique pointers. This promotes code reuse by allowing us to write functions that can operate on any kind of pointer, provided that we obey the restrictions of both unique pointers (restricted sharing) and nonunique pointers (no `free`). Thus, a key contribution of our work is a coherent, full-scale design that incorporates both regions and uniqueness.

Another contribution of this work is that we show how to leverage regions to avoid some of the strong restrictions on unique pointers (or linear types) that have plagued other systems. In particular, regions permit us to define a scoped *alias* construct for temporarily sharing (or "borrowing") a unique pointer.[1] Our *alias* construct, inspired by theoretical work of Walker and Watkins [34], uses a locally-scoped region name and the type-and-effects system to ensure that copies of the pointer do not escape. Without *alias*, many library functions would have to be rewritten to accommodate the constraints of unique pointers.

Previous work (discussed in Section 6) that attempted to leverage unique pointers and provide some sort of borrowing mechanism was forced either to perform global pointer analyses (to ensure that the copies did not escape) or to require steeper restrictions, which would force more libraries to be rewritten. To ensure separate compilation and scalable type-checking, we prefer a more modular approach. Regions provide exactly the needed modularity by giving us a way to name pointers and control where (usable copies of) those pointers may flow. The price paid is that region annotations must be placed on types, and effects must be placed on function signatures. However, the combination of local inference and default annotations minimizes this burden. For instance, when porting C code to region-based Cyclone, we found that on average less than 2% of the lines of code had to be modified with region annotations [17].

We have implemented our extensions in the Cyclone compiler, which is freely available. In our experience, it is very useful to have uniqueness, reference counting, regions, and garbage collection. In particular, unique pointers and reference counted objects provide finer control over object lifetimes, but at the cost of sometimes programming with restricted (alias-free) idioms and/or writing a few more annotations. On the other hand, garbage collection is very easy to use, but provides far less control. Users can choose strategies that work best for their application. Section 5 describes our experience developing MediaNet [23],

[1] We also leverage uniqueness to enrich regions: Using a unique pointer to a region, we can relax some of regions' LIFO restrictions. Unfortunately, space limits preclude a discussion of regions more flexible than in our previous work. Our technical report [22] has details.

an overlay network for streaming data (other applications are described in our Technical Report [22]). With fairly modest changes, we adapted MediaNet to use unique and reference-counted pointers, thereby reducing memory consumption by roughly an order of magnitude and increasing throughput by up to 42%.

2 Cyclone Regions Background

Previous work describes our type system for region-based memory management in detail [17]. Here we discuss only the aspects most crucial to unique pointers.

Cyclone pointer types have the form `t*‘r`, where `t` is the type of the pointed-to object and `‘r` is a *region name* describing the object's lifetime. The type-and-effects system computes a set of "definitely live" regions for each program point. Dereferencing a pointer of type `t*‘r` requires that `‘r` is in the set, making dangling-pointer dereferences a compile-time error. Region polymorphism lets functions and data structures abstract over the regions of their arguments and fields. For example, the following `struct` definition for linked lists abstracts over the elements in the list (`‘a`), and the region of the list cells (`‘r`):

```
struct List<‘a,‘r> {
  ‘a hd;
  struct List<‘a,‘r> *‘r tl;
};
typedef struct List<‘a,‘r> *‘r list_t<‘a,‘r>;
```

Programmers specify the region in which to allocate a new object. A combination of default effects, default region annotations, and local type inference means that programmers rarely have to write region annotations. However, in this paper, we will put in all of the region annotations to clarify the examples.

A key restriction on region deallocation simplifies the effect system so that it is intraprocedural and syntax-directed: If `‘r` is accessible on entry to a scope (e.g., a function body), then it is accessible throughout the scope. This property is natural for stack-allocated objects and lexically-scoped regions, but it can force us to retain regions longer than necessary.

If we can determine statically that `‘r1` *outlives* `‘r2`, then `t*‘r1` is a subtype of `t*‘r2` because it permits access at more program points. Together with the just-described restriction, subtyping means if `‘r1` is accessible when entering a scope that creates and destroys a region named `‘r2`, then `t*‘r1` is a subtype of `t*‘r2`.

3 Unique Pointers

Lexically-scoped regions are not an efficient mechanism for small sets of objects, or for sets of objects that need to be deallocated at widely varying times. However, in the presence of aliases, freeing individual objects can lead to unsafe programs. In particular, calling `free(x)` may deallocate an object referred to by another variable `y`, introducing a dangling pointer. We can avoid this by limiting use of `free` to *unique*—unaliased—pointers.

The idea of using unique pointers is derived from linear and affine type systems, and has been proposed in many settings (see Section 6). This section explains how our design relaxes important limitations of conventional approaches:

- We allow unique pointers within shared objects, and provide an atomic swap operator that lets them be accessed safely.
- We provide additional polymorphism to let us abstract over types that may contain unique pointers or nonunique pointers.
- We provide support for temporarily treating a collection of unique pointers as if they were pointers into a region. Hence we can reuse code for "reader" functions (e.g., computing a list's length) without awkward coding idioms.

3.1 Simple Unique Pointers

A unique pointer can be created by calling `malloc` and destroyed by calling `free`. To distinguish unique pointers from other pointers, we use types of the form `t*‘U`. Here, `‘U` is a distinguished region name that indicates uniqueness. Semantically, we consider `t*‘U` an abbreviation for (∃!`‘r`. `t*‘r`) where we interpret the binding ∃!`‘r` as meaning "there exists a unique 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.

We use an intraprocedural, flow-sensitive, path-insensitive analysis to track when a unique pointer becomes *consumed*, in which case the analysis rejects a subsequent attempt to use it. We chose an intraprocedural analysis to ensure modular checking and a path-insensitive analysis to ensure scalability. To keep the analysis simple, we treat assignments, function calls, and other expressions as consuming any values they copy. This ensures that there is at most one usable alias of a unique pointer at any program point. Here is a small example:

```
int *‘U q = p;  //consumes p
*q = 5;         //q not consumed
free(q);        //consumes q
*p = *q;        //illegal -- p and q already consumed
```

The first assignment aliases and consumes `p`, while the call to `free` consumes `q`.[2] Therefore, attempts to dereference `p` or `q` are illegal. Dereferencing a unique pointer does not consume it since it does not copy the pointer, as the first dereference of `q` shows.

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 not consumed and we write:

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

then the analysis treats `p` as consumed after the conditional; if `p` was not freed, it will be reclaimed by the garbage collector. We can issue a warning in this situation (and a few others such as overwriting a defined unique pointer). This

[2] Interestingly, `free` is simply a normal function in the view of the analysis.

approach works particularly well with exception handlers, which typically have a large number of incoming control-flow edges (at least one for each function call within the scope of the handler); it is almost never the case that the same unique pointers have been consumed on every edge.

Enriching Cyclone with this analysis was natural because Cyclone already used a sound flow analysis with local must points-to information to prevent dereferencing `NULL` pointers or using uninitialized memory [15].

3.2 Unique Pointers in Shared Data

Copies of unique pointers are permitted only along *unique paths*. A unique path u has the form

$$u \texttt{ ::= } x \texttt{ | } u\texttt{.m} \texttt{ | } u\texttt{->m} \texttt{ | } \texttt{*}u$$

where x is a local variable and u is a unique pointer. To appreciate the unique-path restriction, consider this incorrect code:[3]

```
int unsafe(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`.

Many languages based on linear types avoid the above problem by prohibiting unique pointers in nonunique containers: Above, the attempt to initialize `z` with `*y` would be a compile-time error. However, this policy is too restrictive. For instance, we could never store a unique pointer in a global variable.

Our approach permits sharing unique pointers, but allows extracting them only via an atomic *swap* operation, as suggested by Baker [4]. In Cyclone, swap is written e_1 `:=:` e_2, where e_1 and e_2 are left-hand-side expressions of unique-pointer type, including paths that go through nonunique pointers. 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. This idea is formalized in our work on linearly typed assembly language [10] and can also be justified with formalisms such as alias types [28].

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);
}
```

[3] The syntax `int *‘U*‘r` is pronounced, “A `‘r` pointer to a `‘U` pointer to an `int`.”

Aliasability	May have aliases	May create aliases
normal	×	×
unique (U)		
top (T)	×	

normal ≤ top (T)
unique (U) ≤ top (T)

Fig. 1. Kinds and Aliasabilities

Here, g is a global variable holding a unique pointer. The `init` routine creates the unique pointer and stores it in `temp`. Then, the value of the temporary is swapped for the value of g. Afterward, 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, the swap is atomic, so this property holds even if multiple threads were to execute `init` concurrently. Atomic swap can be used to build useful concurrent protocols, while simple extensions, like compare-and-swap, can be used to build arbitrary wait-free structures [21].

An alternative to swap is a *destructive read*. Destructive reads are typically implemented as *implicit* swaps with `NULL`, but this presumes that all pointer types admit `NULL` as a value. Cyclone has a pointer qualifier, `@notnull`, which indicates `NULL` is not allowed. This qualifier helps avoid the costs and failures of potential `NULL`-pointer dereferences. Making swap *explicit* ensures that unique, `@notnull` pointers can still be read out of shared data structures.

3.3 Polymorphism

Cyclone supports polymorphism, which is crucial for writing reusable library functions. With care, we can extend this polymorphism to handle unique pointers. The following function illustrates some of the difficulties. It takes a (nonempty) list and makes it a circular list, and places the list's first element into the location referenced by the second parameter:

```
list_t<‘a,‘r> cycle(list_t<‘a,‘r> x, ‘a *‘r2 h) {
  list_t<‘a,‘r> res = x;
  while (x->tl != NULL) x = x->tl;
  x->tl = res; //creates cycle
  *h = res->hd; //copies first element's value
  return res;
}
```

Circular lists clearly violate our uniqueness invariant, so we do not expect `cycle` to work on lists allocated in `‘U`. In particular, if we replace `‘r` with `‘U`, the function body does not typecheck: `x` is consumed when initializing `res`, so it cannot be used in the while loop. The assignment to `*h` presents a second problem if `‘a` can be instantiated with a unique pointer, since it creates a copy of `res->hd`.

To distinguish between regions (and type variables) that can and cannot be considered unique, we classify types with both an *aliasability* and a *kind*. Kinds describe the structure of types, e.g., `B` describes boxed (i.e., pointer) types, `R` describes regions, etc. Aliasabilities indicate how a type may be aliased, and are

```
list_t<`a::TB,`r::TR> 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;
  }
```

Fig. 2. A polymorphic function for imperatively reversing a list

shown in Figure 1. A *normal* aliasability indicates that a type's objects may have aliases and that aliases can be freely created. In contrast, a *unique* aliasability indicates that a type's objects are not aliased, nor can they ever be. For example, the region `` `U `` has kind `UR` and the type `int *`U` has kind `UB`. Normal aliasability is the default. As such, the type of `cycle` prohibits instantiating either `` `a `` with a unique pointer, or `` `r `` with `` `U ``. That is, `cycle`'s type is:

$\forall$`` `a::B,`r::R,`r2::R.(list_t<`a,`r>,`a *`r2) `` $\rightarrow$ `list_t<`a,`r>`

For more code reuse, we define a third aliasability `T` (for "top"), whose types' objects could have aliases, but for which no new aliases may be created. For instance, a value of type `` `a::TB `` cannot be freely duplicated *and* cannot be considered unique. These conditions permit using unique- or normal-kinded types where we expect top-kinded types, creating the subaliasing relationship shown in Figure 1. This enables writing functions polymorphic over aliasability. For instance, the function `imp_rev` in Figure 2 destructively reverses a list, regardless of the (pointer) type used to instantiate `` `a `` or the kind of region used for `` `r ``.

3.4 Temporary Aliasing

Programmers often write code that creates local copies of pointers that never escape. For example, here is a length function for lists:

```
struct ROList<`a,`r> { // read-only lists
  `a hd;
  struct ROList<`a,`r> *`r const tl;
};
typedef struct ROList<`a,`r> *`r rolist_t<`a,`r>;
int length(rolist_t<`a,`r> x) {
  int i=0;
  for(; x != NULL; x = x->tl) ++i;
  return i;
}
```

The code begins by defining a *read-only* list type `rolist_t`. Cyclone supports a form of subtyping that allows a mutable `list_t` (defined in Section 2) to be used wherever a read-only `rolist_t` is expected. To be general, the `length` function has been written to take read-only lists.

Within `length`, the variable `x` is used to hold copies of the pointers to each list cell, but these copies do not escape from the function. Intuitively, `length` should work on unique and nonunique lists. But for a unique list, the caller would have to assume that the list is consumed, which makes it impossible to use the list after calling `length`. There are other ways to code this function so that it becomes useful for unique lists (e.g., using swaps and returning the input), but this would force us to make the lists mutable, precluding length from being used on immutable lists. In short, it seems difficult to write functions that meet all of the constraints needed to achieve truly reusable code.

Inspired by the theoretical work of Walker and Watkins [34], we address this problem by introducing an `alias` declaration that permits temporary aliasing of unique and top-kinded pointers for the duration of a block. This primitive resembles and extends Walker and Watkins' `let!`, the `unpack` primitive of alias types [28], and Clarke's notion of borrowing [11]. Here is an example:

```
int length_and_free(list_t<`a,`U> x) {
  int len;
  { alias <`r2> rolist_t<`a,`r2> temp = x; //consumes x
    len = length(temp);
  } //unconsumes x
  freelist(x);
  return len;
}
```

The function acquires the length of a unique list, using the generic `length`, and then frees the list before returning the length. The `alias` declaration temporarily aliases `x` by (a) introducing a fresh region variable `` `r2 `` of kind `R` (i.e., normal aliasability) and (b) by copying `x` to the local variable `temp` of type `rolist_t<`a,`r2>`, which can be passed to `length`. The original unique list, `x`, is considered consumed for the duration of the block. Thus, it is impossible for the value to be freed during the execution of the `alias` block. At the end of the block, any copies become unusable, since the region system ensures that `` `r2 `` becomes inaccessible. This allows us to once again treat `x` as if it is a unique list so that we can pass it to `freelist`. In short, the region system ensures copies do not escape without imposing any additional restrictions on `length`.

The flow analysis and type system are preventing the unique pointer from being deallocated, at least temporarily. Thus, if we introduce a fresh region `` `r2 ``, a unique pointer will always *outlive* `` `r2 ``. Thus, it is safe to treat τ `*`U` as a subtype of $\tau$ `*`r2`. In the above example, we are extending this notion to treat an indeterminate number of unique pointers (the `tl` fields of each of the list elements) as if they were references into `` `r2 ``.

Note that such "deep subtyping" is sound only for covariant type constructors (such as `rolist_t`). The following example makes clear the reason why:

```
'a oops(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; //dangling-pointer dereference
}
```

In the example, we overwrite one unique pointer with another to create a circular list by exploiting `alias`. The type-checker would not reject the assignment since we have temporarily given 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. To prevent this problem, we must therefore restrict deep aliasing to read-only paths, as the rules of deep subtyping suggest.

For programmer convenience, the Cyclone typechecker 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`. This backtracking scheme, which is much like Aiken et al.'s approach for inferring uses of a similar `confine` construct [3], allows the `alias` block in `length_and_free` to be inferred:

```
int length_and_free(list_t<'a,'U> x) {
  int len = length(x);
  freelist(x);
  return len;
}
```

In practice, we have found that `alias` drastically reduces the number of functions that we have to write, and allows us to reuse many of the existing, region-based libraries.

4 Reference-Counted Objects

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

We define a new *reference-counted region* `'RC`. Objects allocated in `'RC` are prepended with a hidden reference-count field. We assign `'RC` the kind `TR` to specify that reference-counted pointers might be aliased, but no implicit aliases may be created. Instead, `'RC` pointers must be copied *explicitly* by calling `alias_refptr`, which returns an alias and increases the reference count of the underlying object. It has this type:[4]

[4] Note that `alias_refptr` is treated specially in that its argument is not consumed.

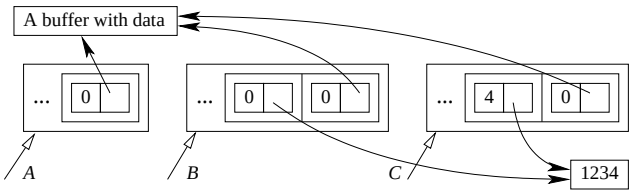

Fig. 3. Pointer graph for three streambuffs

```
‘a *‘RC alias_refptr(‘a *‘RC);
```

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 that a pointer is not prematurely freed due to a mismanaged count. All the polymorphism described earlier also applies. For example, a function like `imp_rev` (Section 3.3) that abstracts over `TR` can be passed a reference-counted object.

5 Programming Experience

Cyclone has been used for several projects where safety and resource management, particularly with memory, are important [27, 26, 7]. We have written two applications, an overlay network called MediaNet [23] and a web server, that employ unique and reference-counted pointers extensively to improve performance. Due to space constraints, we present in detail only MediaNet's construction and performance, concluding with a general assessment of our new constructs. Further details can be found in our technical report [22].

5.1 MediaNet

MediaNet is an overlay network for performing on-line, adaptive scheduling for packet streams with user-specified resource constraints [23]. 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 reconfigurable DAG of *operations*, where each operation works on the data as it passes through.

The packets sent between operations are called *streambuffs*,[5] which are essentially arrays of *databuffs*. A databuff is a pair, consisting of a reference-counted pointer to a buffer of bytes, and an integer offset within that buffer.

Streambuffs are allocated uniquely, so they can be freed immediately after the corresponding data is sent. When multiple streambuffs must refer to portions of

[5] Streambuffs are a simpler variant of Linux's `skbuff`s.

the same packet data, we clone them as shown in Figure 3. 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 streambuff 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.

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. As the current version uses `‘RC` pointers instead, the possibility of mismanaging the count is greatly reduced, and data is freed immediately after its last use.

Moving streambuffs and databuffs to unique pointers and reference counting significantly improved performance. In a simple experiment, we used the TTCP microbenchmark [25] 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 [6], version 6.2α4, for garbage collection and manual deallocation. Cyclone frequently interacts with C, and so cannot use a more precise collector.
- *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 4 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 semi-interquartile range[6] 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%.

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

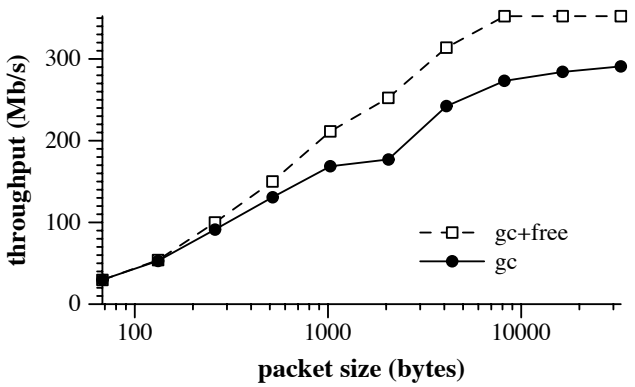

Fig. 4. MediaNet throughput

For 4 KB packets, the `gc` configuration reserves roughly 820 KB of memory for memory management, with a peak memory usage of 420 KB, garbage collecting a total of 80 times. In contrast, the `gc+free` configuration reserves 128 KB, with a peak usage of 8 KB, and never garbage collects. This usage corresponds to some initial data and memory for a single packet, which is always freed immediately after it is sent. Using the Lea allocator [24] for `gc+free` reduces the amount of reserved space (25 KB); the BDW collector apparently has a high rate of fragmentation. These trends are roughly the same for other packet sizes.

5.2 Overall Experience

In our experience porting C code to Cyclone code that uses GC, roughly 10% of the original program needs minor adjustments, and only about 2% of the total changes are region-related [17]. Porting libraries typically requires more annotations than application code. While the basic trends are the same, moving programs to use unique pointers rather than GC typically requires more work.

First, type variables appearing in function prototypes assume normal aliasability unless otherwise specified; thus a function over arbitrary regions requires an explicit annotation. Second, to take advantage of the `alias` construct, we may need to define read-only versions of data structures, and adjust reader functions to use those versions instead, as we did with list `length` in Section 3.4. Alias inference reduces changes to clients. Both of these sorts of changes do not modify code, but rather only add small annotations or define new types.

Code that constructs or alters datastructures must often be modified to work for arbitrary regions, e.g., by using swaps, as shown for the `imp_rev` function in Figure 2. To avoid the added performance overhead of the general function, one could retain versions for aliasable regions only. For the list library, doing this would result in 22 new functions, out of 80 total. Most challenging is writing containers for generic non-aliasable values, because we must use swap to access or free those values in a potentially application-specific way.

Region Variety	Allocation (objects)	Deallocation (what)	Deallocation (when)	Aliasing (objects)
Stack	static	whole region	exit of lexical scope	unrestricted
Lexical	dynamic			
Heap (`H)			automatic (GC)	
Unique (`U)		single objects	manual	restricted
Reference-counted (`RC)				

Fig. 5. Summary of Cyclone Regions

Putting this all together, our experience with MediaNet was positive. Generalizing streambuffs to arbitrary regions, and moving databuffs to `RC pointers, resulted in less than 20% of the streambuff library being changed, and less than 5% of the code overall. Out of 13000 lines total, we added 76 annotations, used swap 46 times, and `alias` 66 times, of which 71% were automatically inferred. This can be improved. For example, a more general constraint-based inference should be able to discover many of the `alias` statements not already inferred. Similarly, support for a `restrict` mechanism in the style of Aiken et al. [3] should reduce the need for swapping, at the cost of more annotations.

Figure 5 summarizes Cyclone's memory management system. It illustrates the flexibility that programmers have, particularly in choosing how to deallocate objects. As the alias restrictions indicate, a particular choice essentially trades ease of use for better control over object lifetime.

6 Related Work

The ML Kit [29] implements Standard ML with regions. Whole-program analysis (type inference) assigns regions using a system that (like Cyclone) has lexically-scoped regions as its backbone [30]. Various extensions have relaxed some LIFO restrictions [2, 18], but unique pointers have not been considered.

The RC language and compiler [14] 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 most 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.

Work on linear types [31], alias types [28, 33], capabilities [32], and linear regions [34, 20] 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.

Vault [12, 13] is another project adapting 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 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. Their 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 [16] remains future work.

Unique pointers and related alias restrictions have received considerable attention as extensions to object-oriented languages. Clarke and Wrigstad [11] provide an excellent review of related work and propose a notion of "external uniqueness" that integrates unique pointers and ownership types. Prior to this work, none of the analogues to Cyclone's *alias* allowed aliased pointers to be stored anywhere except in method parameters and local variables, severely restricting code reuse. Clarke and Wrigstad use a "fresh owner" to restrict the escape of aliased pointers, much as Cyclone uses a fresh region name with *alias*, but they "borrow" (i.e., alias) only one object. Ownership types differ from our region system in several ways, most notably by restricting what objects can refer to what other objects instead of using a static notion of accessible regions at a program point.

Little work on uniqueness in OO languages has targeted manual memory management. A recent exception is Boyapati et al.'s work [8], which uses regions to avoid some run-time errors in Real-Time Java programs. As is common, this work always requires "destructive reads" (an atomic swap with `NULL`) and relies on an optimizer to eliminate unnecessary writes of `NULL` on unique paths. Cyclone resorts to swaps only for unique data in nonunique containers, catching more errors at compile time. Few other projects have used swap instead of destructive reads [4, 19]. Alias burying [9] eschews destructive reads and proposes using static analysis to prevent using aliases after a unique pointer is consumed, but the details of integrating an analysis into a language definition are not considered.

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* [5] 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.

7 Conclusions

Cyclone now supports a rich set of safe memory-management idioms:

– *Stack/regions:* We can avoid any run-time cost for data whose lifetime is known sufficiently well when allocated.
– *Heap:* We allow conservative garbage collection for some 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.

Programmers 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. Regions are the backbone of our system and exploit the convenience of data accessibility corresponding to scope. We regain this convenience for unique and reference-counted pointers with `alias`, which 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 can be awkward. Finally, we provide run-time checking when static enforcement is too onerous: 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.

References

1. Peter Achten and Rinus Plasmeijer. The ins and outs of Clean I/O. *Journal of Functional Programming*, 5(1), 1995.
2. Alex Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *PLDI*, 1995.
3. Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *PLDI*, 2003.
4. Henry Baker. Lively linear LISP—look ma, no garbage. *ACM SIGPLAN Notices*, 27(8), 1992.
5. Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *OOPSLA*, 2002.
6. Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software – Practice and Experience*, 18(9), 1988.
7. Herbert Bos and Bart Samwel. Safe kernel programming in the OKE. In *IEEE OPENARCH*, 2002.
8. Chandrasekhar Boyapati, Alexandru Sălcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time Java. In *PLDI*, 2003.
9. John Boyland. Alias burying: Unique variables without destructive reads. *Software – Practice and Experience*, 31(6), 2001.
10. James Cheney and Greg Morrisett. A linearly typed assembly language. Technical Report 2003-1900, Department of Computer Science, Cornell University, 2003.

11. Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP*, 2003.
12. Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, 2001.
13. Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, 2002.
14. David Gay and Alex Aiken. Language support for regions. In *PLDI*, 2001.
15. Dan Grossman. *Safe Programming at the C Level of Abstraction*. PhD thesis, Cornell University, 2003.
16. Dan Grossman. Type-safe multithreading in Cyclone. In *TLDI*, 2003.
17. Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *PLDI*, 2002.
18. Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *PLDI*, 2002.
19. Douglas Harms and Bruce Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5), 1991.
20. Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In *PPDP*, 2001.
21. Maurice Herlihy. Wait-free synchronization. *TOPLAS*, 13(1), 1991.
22. Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe and flexible memory management in Cyclone. Technical Report CS-TR-4514, University of Maryland Department of Computer Science, July 2003.
23. Michael Hicks, Adithya Nagajaran, and Robbert van Renesse. MediaNet: User-defined adaptive scheduling for streaming data. In *IEEE OPENARCH*, 2003.
24. Doug Lea. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`.
25. Mike Muuss. The story of TTCP. `http://ftp.arl.mil/~mike/ttcp.html`.
26. Parveen Patel and Jay Lepreau. Hybrid resource control of active extensions. In *IEEE OPENARCH*, 2003.
27. Parveen Patel, Andrew Whitaker, David Wetherall, Jay Lepreau, and Tim Stack. Upgrading transport protocols using untrusted mobile code. In *SOSP*, 2003.
28. Fred Smith, David Walker, and Greg Morrisett. Alias types. In *ESOP*, 2000.
29. Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions in the ML Kit (for version 4). Technical report, IT University of Copenhagen, 2001.
30. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2), 1997.
31. Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990. IFIP TC 2 Working Conference.
32. David Walker, Karl Crary, and Greg Morrisett. Typed memory management in a calculus of capabilities. *TOPLAS*, 24(4), 2000.
33. David Walker and Greg Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, 2000.
34. David Walker and Kevin Watkins. On regions and linear types. In *ICFP*, 2001.