

# Region-Based Memory Management in Cyclone

Dan Grossman      Greg Morrisett      Trevor Jim<sup>†</sup>  
Michael Hicks      Yanling Wang      James Cheney

Computer Science Department  
Cornell University  
Ithaca, NY 14853  
{danieljg,jgm,mhicks,wangyl,jcheney}@cs.cornell.edu

<sup>†</sup>AT&T Labs Research  
180 Park Avenue  
Florham Park, NJ 07932  
trevor@research.att.com

## ABSTRACT

Cyclone is a polymorphic, type-safe programming language derived from C. The primary design goals of Cyclone are to let programmers control data representations and memory management without sacrificing type-safety. In this paper, we focus on the region-based memory management of Cyclone and its static typing discipline. The design incorporates several advancements, including support for region subtyping and a coherent integration with stack allocation and a garbage collector. To support separate compilation, Cyclone requires programmers to write some explicit region annotations, but uses a combination of default annotations, local type inference, and a novel treatment of region effects to reduce this burden. As a result, we integrate C idioms in a region-based framework. In our experience, porting legacy C to Cyclone has required altering about 8% of the code; of the changes, only 6% (of the 8%) were region annotations.

## 1. INTRODUCTION

Many software systems, including operating systems, device drivers, file servers, and databases require fine-grained control over data representation (e.g., field layout) and resource management (e.g., memory management). The *de facto* language for coding such systems is C. However, in providing low-level control, C admits a wide class of dangerous — and extremely common — safety violations, such as incorrect type casts, buffer overruns, dangling-pointer dereferences, and space leaks. As a result, building large systems in C, especially ones including third-party extensions, is perilous. Higher-level, type-safe languages avoid these drawbacks, but in so doing, they often fail to give programmers the control needed in low-level systems. Moreover, porting or extending legacy code is often prohibitively expensive. Therefore, a safe language at the C level of abstraction, with an easy porting path, would be an attractive option.

Toward this end, we have developed *Cyclone* [6], a lan-

guage designed to be extremely close to C while remaining type-safe. We have written or ported over 70,000 lines of Cyclone code, including the Cyclone compiler, an extensive library, lexer and parser generators, compression utilities, a Windows device driver, and a web server. In so doing, we identified many common C idioms that are usually safe, but for which the C type system is too weak to verify. We then augmented the language with modern features and typing technologies so that programmers could continue to use those idioms, but have safety guarantees.

For example, to reduce the need for type casts, Cyclone has features like parametric polymorphism, subtyping, and tagged unions. To prevent bounds violations without making hidden data-representation changes, Cyclone has a variety of pointer types with different compile-time invariants and associated run-time checks. Other projects aimed at making legacy C code safe have addressed these issues with somewhat different approaches, as discussed in Section 7.

In this paper, we focus on the most novel aspect of Cyclone: its system for preventing dangling-pointer dereferences and space leaks. The design addresses several seemingly conflicting goals. Specifically, the system is:

- *Sound*: Programs never dereference dangling pointers.
- *Static*: Dereferencing a dangling pointer is a compile-time error. No run-time checks are needed to determine if memory has been deallocated.
- *Convenient*: We minimize the need for explicit programmer annotations while supporting many C idioms. In particular, C code that manipulates stack pointers often requires no modification.
- *Exposed*: We provide mechanisms that let programmers control the placement and lifetimes of objects. As in C, local declarations are always stack-allocated.
- *Comprehensive*: We treat all memory uniformly, including the stack, the heap (which can be optionally garbage-collected), and “growable” regions.
- *Scalable*: The system supports separate compilation, as all analyses are intra-procedural.

Following the seminal work of Tofte and Talpin [22], the system is *region-based*: each object lives in a distinct region of memory and, with the optional exception of the heap, a region’s objects are all deallocated simultaneously. As a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.  
Copyright 2001 ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

static system for an explicitly typed, low-level language, Cyclone’s region framework makes several important technical contributions over previous work, notably:

- Region subtyping: A last-in-first-out discipline on region lifetimes induces an “outlives” relationship on regions, which, in turn, allows us to provide a useful subtyping discipline on pointer types.
- Simple effects: We eliminate the need for effect variables (which complicate interfaces) through the use of a “`regions_of`” type operator.
- Local region inference: Though inference is local, a system of defaults minimizes the need for explicit region annotations.
- Integration of existentials: The combination of region subtyping and simple effects makes the integration of first-class abstract data types relatively simple.

In the rest of this paper, we demonstrate these contributions. We begin with a general description of the system suitable for C programmers (Section 2), and then follow with a more technical discussion of our novel effect system and its interaction with existential types (Section 3). We continue with a core formal language that we have proven sound (Section 4), an overview of our implementation (Section 5), and a study of the cost of porting C code to Cyclone (Section 6). We discuss related work in Section 7 and future work in Section 8.

## 2. USING CYCLONE REGIONS

This section presents the programmer’s view of Cyclone’s memory-management system. It starts with the constructs for creating regions, allocating objects, and so on — this part is simple because the departure from C is small. We next present the corresponding type system, which is more involved because every pointer type carries a region annotation. Then we show how regions’ lifetimes induce subtyping on pointer types. At that point, the type syntax is quite verbose, so we explain the features that, in practice, eliminate almost all region annotations. Throughout, we take the liberty of using prettier syntax (e.g., Greek letters) than actual Cyclone. For the ASCII syntax and a less region-oriented introduction to Cyclone, see the user’s manual [6].

### 2.1 Basic Operations

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

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

Static data are placed in the heap. Primitives `malloc` and `new` create new heap objects. The `new` operation is like `malloc` except that it takes an expression and initializes the memory with it. There is no explicit mechanism for reclaiming heap-allocated objects (e.g., `free`). However, Cyclone programs may optionally link against the Boehm-Demers-Weiser conservative garbage collector [4] to implicitly reclaim unreachable heap-allocated objects. The interaction of the collector with regions is discussed in Section 5.

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

Dynamic regions are created with the construct `region r {s}`, where `r` is an identifier and `s` is a statement. The region’s lifetime is the execution of `s`. In `s`, `r` is bound to a *handle* for the region, which primitives `rmalloc` and `rnew` use to allocate objects into the associated region. For example, `rnew(r) 3` returns a pointer to an `int` allocated in the region of handle `r` and initialized to 3. Handles are first-class values; a caller may pass a handle to a function to allow it to allocate into the associated region. A pre-defined constant `heap_region` is a handle for the heap.

Like a declaration block, a dynamic region is deallocated precisely when execution leaves the body of the enclosed statement. Execution can leave due to unstructured jumps (`continue`, `goto`, etc.), a `return`, or via an exception. Section 5 explains how we compile dynamic-region deallocation.

The region system imposes no changes on the representation of pointers or the meaning of operators such as `&` and `*`. There are no hidden fields or reference counts for maintaining region information at run-time. Pointers to arrays of unknown size (denoted  $\tau ?$ ) are implemented with extra fields to support bounds-checks, but this design is orthogonal to regions. As a result, all the infrastructure for preventing dangling-pointer dereferences is in the static type system, making such dereferences a compile-time error.

### 2.2 Basic Type System

*Region Annotations.* All pointers point into exactly one region. In principle, pointer types are annotated with the *region name* of the region they point into, though in practice we eliminate most annotations. Ignoring subtyping, `int* $\rho$`  describes a pointer to an `int` that is in the region whose name is  $\rho$ . This invariant—pointers have a particular region—is the basic restriction we impose to make the undecidable problem of detecting dangling-pointer dereferences tractable. Pointer types with different region names are different types. A handle for a region corresponding to  $\rho$  has the type `region_t< $\rho$ >`.

Region names fall into four categories. The region name for the heap is  $\rho_H$ . A block labeled `L` (such as `L:{int x=0; s}`) has name  $\rho_L$ , and refers to the stack region that the block creates. Similarly, the arguments of a function `f` are stored in the stack region  $\rho_f$ . Finally, the statement `region r {s}` defines region name  $\rho_r$  for the created region. So `r` has type `region_t< $\rho_r$ >`. In all cases, the scope of a region name corresponds to the lifetime of the corresponding region.

We can now give types to some small examples. If `e1` has type `region_t< $\rho$ >` and `e2` has type  $\tau$ , then `rnew (e1) e2` has type  $\tau*\rho$ . If `int x` is declared in block `L`, then `&x` has type `int* $\rho_L$` . Similarly, if `e` has type  $\tau*\rho$ , then `&*e` has type  $\tau*\rho$ .

*Preventing dangling-pointer dereferences.* To dereference a pointer, safety demands that its region be live. Our goal is to determine at compile-time that no code follows a dangling pointer. It appears that no well-typed pointer could be a dangling reference, because pointer types’ region

```

char?ρ strcpy<ρ,ρ₂>(char?ρ d, const char?ρ₂ s);
char?ρ_H strdup<ρ>(const char?ρ s);
char?ρ rstrdup<ρ,ρ₂>(region_t<ρ>,const char?ρ₂ s);
size_t strlen<ρ>(const char?ρ s);

```

Figure 1: Cyclone string library prototypes

names must be in scope. For example, this code is ill-typed:

```

1. int*ρ_L p;
2. L:{ int x = 0;
3.   p = &x;
4. }
5. *p = 42;

```

The code creates storage for `x` at line 2 and deallocates it at line 4, so the assignment of `&x` to `p` creates a dangling pointer that is dereferenced in line 5. Cyclone rejects this code because  $\rho_L$  is not in scope when `p` is declared. If we change the declaration of `p` to another region, then the assignment `p = &x` fails to type-check because `&x` has type  $\text{int}*\rho_L$ .

However, Cyclone’s advanced features, notably existential and universal polymorphism, conspire to allow pointers to escape the scope of their regions, just as closures allow pointers to escape in the original Tofte-Talpin work. Therefore, in general, we cannot rely upon simple scoping mechanisms to ensure soundness, and must instead track the set of live regions at each control-flow point. To keep the analysis intra-procedural, we use a novel type-and-effects system to track inter-procedural liveness requirements. We delay the full discussion of effects until Section 3.

**Region Polymorphism.** Functions in Cyclone are *region-polymorphic*; they can abstract the actual regions of their arguments or results. That way, functions can manipulate pointers regardless of the region they point into, whether it be the stack, the heap, or a dynamic region.

Figure 1 presents some prototypes from the Cyclone string library, including `strcpy`, `strdup`, and `strlen`, and a region-allocating function `rstrdup`. The `?` is Cyclone notation for a pointer to a dynamically-sized array. These functions all exhibit region polymorphism. In `strcpy`, the parameters’ region names  $\rho$  and  $\rho_2$  are abstracted by the syntax  $\langle\rho, \rho_2\rangle$ , meaning they can be instantiated with any actual region name when the function is called. So we can write code like:

```

L:{ char buf [20];
   strcpy<ρ_L, ρ_H>(buf, "a heap pointer"); }

```

Here, the call instantiates  $\rho_2$  with the heap region  $\rho_H$  and  $\rho$  with the stack region  $\rho_L$ , allowing one to copy a string from the heap to the stack.

Region polymorphism can also guarantee region equalities of unknown regions by using the same region names. For example, in `strcpy` the region names of the first argument and the return value are the same; so the returned pointer must point to the same region as the first argument. Region name equalities are also important for dynamic regions. For example, the `rstrdup` function is a version of `strdup` that copies the source string into a dynamic region. In its prototype, we see that the region name of the returned value  $\rho$  matches the region name of the dynamic region handle `region_t<ρ>`. In fact, we implement `strdup` by just calling `rstrdup`:

```

char?ρ_H strdup<ρ>(const char?ρ s) {
   return rstrdup<ρ_H, ρ>(heap_region,s);
}

```

**Polymorphic Recursion.** It is often valuable to instantiate the region parameters of a recursive function call with different names than the function’s own arguments. As an example, this contrived program has a function `fact` that abstracts a region  $\rho$  and takes as arguments a pointer into  $\rho$  and an integer.

```

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

```

When executed, the program returns the value 720. In `main`, we pass `fact` a heap pointer (`&g`), so the type of `fact` is instantiated with  $\rho_H$  for  $\rho$ . In contrast, the recursive call instantiates  $\rho$  with  $\rho_L$ , which is the name of the stack region. At run time, the first instance of `fact` modifies `g`; each recursive call modifies the value of `x` in its caller’s stack frame.

**Type Definitions.** Because `struct` definitions can contain pointers, Cyclone allows `structs` to be parameterized by region names. For example, here is a type for lists of pointers to ints:

```

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

```

Ignoring subtyping, a value of type `struct Lst<ρ₁, ρ₂>` will be a list with `hd` fields that point into  $\rho_1$  and `tl` fields that point into  $\rho_2$ . Other invariants are possible: If the type of `tl` were `struct Lst<ρ₂, ρ₁>* ρ₂`, we would describe lists where the regions for `hd` and `tl` alternated at each element.

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

```

typedef struct Lst<ρ_H, ρ> list_t<ρ>;

```

## 2.3 Subtyping

While the type system we have described thus far is quite powerful, it is not expressive enough in some cases. For example, it is common to define a local variable to alternatively hold the value of one of its arguments:

```

void f<ρ₁, ρ₂>(int b, int*ρ₁ p1, int*ρ₂ p2) {
   L: { int*ρ_L p;
       if(b) p = p1; else p=p2;
       /* ... do something with p ... */ }
}

```

In the type system described thus far, the program fails to type-check because neither `p1` nor `p2` has type  $\text{int}*\rho_L$ . We cannot change the type of `p` to  $\text{int}*\rho_1$  or  $\text{int}*\rho_2$ , for then one of the assignments would fail.

To solve this problem, we observe that if the region corresponding to  $\rho_1$  *outlives* the region corresponding to  $\rho_2$ ,

```

char? $\rho$  strcpy(char? $\rho$  d, const char? s);
char? strdup(const char? s);
char? $\rho$  rstrdup(region_t< $\rho$ >, const char? s);
size_t strlen(const char? s);

```

Figure 2: Cyclone prototypes minimally-annotated

then it is sound to use a value of type  $\tau * \rho_1$  where we expect one of type  $\tau * \rho_2$ . Cyclone supports such coercions implicitly. The last-in-first-out region discipline makes such outlives relationships common: when we create a region, we know every region currently alive will outlive it. Simple subtyping based on this outlives relationship allows the above program to type-check.

Region-polymorphic functions can specify outlives relationships among their arguments with explicit pre-conditions that express partial orders on region lifetimes. In practice, we have not used this feature, because the local outlives information has sufficed.

To ensure soundness, we do not allow casting  $\tau_1 * \rho$  to  $\tau_2 * \rho$ , even if  $\tau_1$  is a subtype of  $\tau_2$ , as this cast would allow putting a  $\tau_2$  in a location where other code expects a  $\tau_1$ . (This problem is the usual one with covariant subtyping on references.) However, Cyclone does allow casts from  $\tau_1 * \rho$  to `const`  $\tau_2 * \rho_2$  when  $\tau_1$  is a subtype of  $\tau_2$ . To ensure soundness, we must enforce read-only access for `const` values (unlike C). This support for “deep” subtyping, when combined with polymorphic recursion, is powerful enough to allow stack allocation of some recursive structures of arbitrary size.

## 2.4 Eliminating Annotations

Although Cyclone is explicitly typed in principle, it would be too onerous to fully annotate every function. Instead, we use a combination of inference and well-chosen defaults to dramatically reduce the number of annotations needed in practice. We emphasize that our approach to inference is purely intra-procedural and that prototypes for functions are never inferred. Rather, we use a default completion of partial prototypes to minimize region annotations. This approach permits separate compilation.

When writing a pointer type (e.g., `int*`), the region annotation is optional; the compiler deduces an appropriate annotation based on context:

1. For local declarations, a unification-based inference engine infers the annotation from the declaration’s (intra-procedural) uses. This local inference works well in practice, especially when declarations have initializers.
2. Omitted region names in argument types are filled in with fresh region names that are implicitly generalized. So by default, functions are region polymorphic without any region equalities.
3. In all other contexts (return types, globals, type definitions), omitted region names are filled in with  $\rho_H$  (i.e., the heap). This default works well for global variables and for functions that return heap-allocated results. However, it fails for functions like `strcpy` that return one of their parameters. On the other hand, without looking at the function body, we cannot determine which parameter (or component of a parameter) the function might return.

In addition, when calling a region-polymorphic function, the programmer can omit the explicit region-name instantiation and the inference engine discovers it. As a result of these devices, our `fact` example can become annotation-free:

```

void fact(int* result, int n) {
    int x = 1;
    if(n > 1) fact(&x, n-1);
    *result = x*n;
}

```

Taken another way, the function above, when treated as C code, ports to Cyclone with no additional annotations. Figure 2 shows the same string library functions as Figure 1, but minimally annotated. In all cases, the lack of a region annotation on the argument `s` means the type-checker would insert a fresh region name for the pointer type, and generalize it. The lack of an annotation on the return type of `strdup` defaults to the heap. In total, five region annotations were removed and all generalization became implicit.

While the default annotations and inference engine reduce the burden on the programmer and make porting easier, it is still necessary to put in some explicit annotations to express equalities necessary for safety. For example, if we write:

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

then the code elaborates to:

```
void f2< $\rho_1, \rho_2, \rho_3$ >(int * $\rho_1 * \rho_2$  pp, int * $\rho_3$  p) { *pp=p; }
```

which fails to type-check because  $\text{int} * \rho_1 \neq \text{int} * \rho_3$ . The programmer must insert an explicit region annotation to assert an appropriate equality relation on the parameters:

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

Finally, we employ another technique that dramatically reduces annotations in practice, with regard to type definitions. We can partially apply parameterized type definitions; elided arguments are filled in via the same rules used for pointer types. Here is an aggressive use of this feature:

```

typedef struct Lst< $\rho_1, \rho_2$ > * $\rho_2$  l_t< $\rho_1, \rho_2$ >;
l_t heap_copy(l_t l) {
    l_t ans = NULL;
    for(l_t l2 = l; l2 != NULL; l2 = l2->t1)
        ans = new Lst(new *l2->hd, ans);
    return ans;
}

```

Because of defaults, the parameter type is  $\text{l\_t} < \rho_1, \rho_2 >$  and the return type is  $\text{l\_t} < \rho_H, \rho_H >$ . Because of inference, the compiler assigns `ans` the type  $\text{l\_t} < \rho_H, \rho_H >$  and `l2` the type  $\text{l\_t} < \rho_1, \rho_2 >$ .

## 3. EFFECTS

We argued in Section 2.2 that the scope restrictions on region types prevent pointers from escaping the scope of their region. In particular, a function or block cannot return or assign a value of type  $\tau * \rho$  outside the scope of  $\rho$ ’s definition, simply because you cannot write down a (well-formed) type for the result. Indeed, if Cyclone had no mechanisms for type abstraction, this property would hold.

But if there is some way to hide a pointer’s type in a result value, then the pointer could escape the scope of its

region. For instance, if Cyclone had (upwards-escaping) closures, then one could hide a pointer to a local variable in the closure’s environment, and return the closure outside the scope of the variable, thereby introducing a dangling pointer. This, in and of itself, is not a problem, but if the closure is later invoked, then it might dereference the dangling pointer. This is the critical problem that Tofte and Talpin addresses for functional languages.

Cyclone does not have closures, but it has other typing constructs that hide regions. In particular, Cyclone provides existential types [17], which suffice to encode closures [16] and simple forms of objects [5]. Therefore, it is possible in Cyclone for pointers to escape the scope of their regions.

To address this problem, the Cyclone type system keeps track of the subset of regions that are live at any control-flow point. Following Walker, Crary, and Morrisett [23], we call the set of live regions the *capability*. Before dereferencing a pointer, the type system ensures that the associated region is in the capability. Similarly, before calling a function, Cyclone ensures that regions the function might access are all in the current capability. To this end, function types carry an *effect* that records the set of regions the function might access. The idea of using effects to ensure soundness is due to Tofte and Talpin (hereafter TT). However, our treatment of effects differs substantially from previous work.

The first major departure from TT is that we calculate default effects from the function prototype alone (instead of inferring them from the function body) in order to preserve separate compilation. The default effect is simply the set of regions that appear in the argument or result types. For instance, given the prototype:

```
int* $\rho_1$  f(int*, int* $\rho_1$ *);
```

which elaborates to:

```
int* $\rho_1$  f( $\rho_1, \rho_2, \rho_3$ )<(int* $\rho_2$ , int* $\rho_1$ * $\rho_3$ );
```

the default effect is  $\{\rho_1, \rho_2, \rho_3\}$ . In the absence of polymorphism, this default effect is a conservative bound on the regions the function might access. As with region names in prototypes, the programmer can override the default with an explicit effect. For example, if **f** never dereferences its first argument, we can strengthen its prototype by adding an explicit effect as follows:

```
int* $\rho_1$  f(int* $\rho_2$ , int* $\rho_1$ * $\rho_3$ ;  $\{\rho_1, \rho_3\}$ );
```

In practice, we have found that default effects extremely useful. Indeed, for the 70,000 lines of Cyclone code we have written thus far, we have written one non-default effect.

The second major departure from TT is that we do not have *effect variables*. Effect variables are used by TT for three purposes: (1) to simulate subtyping in a unification-based inference framework, (2) to abstract the set of regions that a closure might need to access, and (3) to abstract the set of regions hidden by an abstract type.

In our original Cyclone design, we tried to use TT-style effect variables. However, we found that the approach does not work well in an explicitly-typed language for two reasons. First, the effect variables introduced by TT to support effect subtyping could only occur free in one location, and all effect variables had to be prenex quantified [20]. Their unification algorithm depended crucially upon these structural invariants. In an explicitly-typed language, we found that

enforcing these constraints was difficult. Furthermore, the prenex quantification restriction prevented first-class polymorphic functions, which Cyclone supports.

Second, we found that effect variables appeared in some library interfaces, making the libraries harder to understand and use. Consider, for instance, an implementation of polymorphic sets:

```
struct Set< $\alpha, \rho, \epsilon$ > {
  list_t< $\alpha, \rho$ > elts;
  int (*cmp)( $\alpha, \alpha$ ;  $\epsilon$ );
}
```

A **Set** consists of a list of  $\alpha$  elements, with the spine of the list in region  $\rho$ . We do not know where the elements are allocated until we instantiate  $\alpha$ . The comparison function **cmp** is used to determine set membership. Because the type of the elements is not yet known, the type of the **cmp** function must use an effect variable  $\epsilon$  to abstract the set of regions that it might access when comparing the two  $\alpha$  values. And this effect variable, like the type and region variable, must be abstracted by the **Set** structure.

Now the library might export the **Set** structure to clients abstractly (i.e., without revealing its definition):

```
struct Set< $\alpha, \rho, \epsilon$ >;
```

The client must somehow discern the connection between  $\alpha$  and  $\epsilon$ , namely that  $\epsilon$  is meant to abstract the set of regions within  $\alpha$  that the hidden comparison function might access.

To simplify the system while solving the problems that effect variables solve, we use a type operator, **regions\_of**( $\tau$ ). This novel operator is just part of the type system; it does not exist at run-time. Intuitively, **regions\_of**( $\tau$ ) represents the set of regions that occur free in  $\tau$ . In particular:

```
regions_of(int) =  $\emptyset$ 
regions_of( $\tau^*\rho$ ) =  $\{\rho\} \cup$  regions_of( $\tau$ )
regions_of( $(\tau_1, \dots, \tau_n) \rightarrow \tau$ ) =
  regions_of( $\tau_1$ )  $\cup \dots \cup$  regions_of( $\tau_n$ )  $\cup$  regions_of( $\tau$ )
```

For type variables, **regions\_of**( $\alpha$ ) is treated as an abstract set of region variables, much like effect variables. For example, **regions\_of**( $\alpha^*\rho$ ) =  $\{\rho\} \cup$  **regions\_of**( $\alpha$ ).

With the addition of **regions\_of**, we can rewrite the **Set** example as follows:

```
struct Set< $\alpha, \rho$ > {
  list_t< $\alpha, \rho$ > elts;
  int (*cmp)( $\alpha, \alpha$ ; regions_of( $\alpha$ ));
}
```

Now the connection between the type parameter  $\alpha$  and the comparison function’s effect is apparent, and the data structure no longer needs to be parameterized by an effect variable. Moreover, **regions\_of**( $\alpha$ ) is the default effect for **int** (\***cmp**)( $\alpha, \alpha$ ), so we need not write it.

Now suppose we wish to build a **Set**<int\* $\rho_1, \rho_2$ > value using some pre-defined comparison function:

```
int cmp_ptr< $\rho_1$ >(int* $\rho_1$  p1, int* $\rho_1$  p2) {
  return (*p1) == (*p2);
}
Set<int* $\rho_1, \rho_2$ > build_set(list_t<int* $\rho_1, \rho_2$ > e) {
  return Set{.elts = e, .cmp = cmp_ptr< $\rho_1$ >};
}
```

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

In addition, unifying function types becomes somewhat easier when default effects are used because, given the same argument and result types, two functions always have the same default effect.

### 3.1 Interaction with Existentials

As mentioned above, Cyclone supports *existential types*, which allow programmers to encode closures. For example, we can give a type for “call-backs” that return an `int`:

```
struct IntFn  $\exists\alpha$  { int (*func)( $\alpha$  env);  $\alpha$  env;};
```

Here, the call-back consists of a function pointer and some abstracted state that should be passed to the function. The  $\alpha$  is existentially bound: Various objects of type `struct IntFn` can instantiate  $\alpha$  differently. When a `struct IntFn` object is created, the type-checker ensures there is a type for  $\alpha$  such that the fields are correctly initialized.

To access the fields of an existential object, we need to “open” them by giving a name to the bound type variable. For example, we can write (in admittedly alien syntax):

```
int apply_intfn(struct IntFn pkg) {
  let IntFn< $\beta$ >{.func = f, .env = y} = pkg;
  return f(y);
}
```

The `let` form binds `f` to `pkg.func` with type `int (*)( $\beta$ )` and `y` to `pkg.env` with type  $\beta$ . So the function call appears well-typed. However, the effect for `f` is `regions_of( $\beta$ )` and we have no evidence that these regions are still live, even though  $\beta$  is in scope. Indeed, the regions may not be live as the following code demonstrates:

```
int read< $\rho$ >(int* $\rho$  x) { return *x; }
struct IntFn dangle() {
  L:{int x = 0;
    struct IntFn ans =
      <int* $\rho_L$ >{.func = read< $\rho_L$ >, .env = &x};
    return ans; }
}
```

Here, the abstracted type  $\alpha$  is instantiated with `int* $\rho_L$`  because the call-back’s environment is a pointer to an `int x` in region  $\rho_L$ . The function for the call-back just dereferences the pointer it is passed. When packaged as an existential, the `int* $\rho_L$`  is hidden and thus the result is well-typed despite the fact that the call-back has a dangling pointer.

In short, to use `struct IntFn` objects, we must “leak” enough information to prove a call is safe. We accomplish this by giving `regions_of( $\alpha$ )` a *bound*:

```
struct IntFn< $\rho$ >  $\exists\alpha$ : $\rho$  { ... };
```

The bound means `regions_of( $\alpha$ )` must all *outlive*  $\rho$ . Therefore, if `pkg` has type `struct IntFn< $\rho$ >`, then we can call the `func` field so long as  $\rho$  is live. In practice, this soundly reduces the “effect” of the call-back to a single region.

## 4. FORMAL SOUNDNESS

In a separate technical report [13], we have defined an operational model of core Cyclone, formalized the type system, and proven type soundness. Space constraints prevent us from duplicating the material here, so we summarize the salient details.

The core includes all of the features relevant to memory management, including stack allocation, dynamic region allocation, polymorphism, and existentials. The operational semantics is a small-step, deterministic rewriting relation ( $\rightarrow$ ) from machine states to machine states. A machine state is a triple  $(G, S, s)$  consisting of a garbage stack  $G$ , a stack  $S$ , and a statement  $s$ . The stacks are finite maps from region names ( $\rho$ ) to regions ( $R$ ), which in turn are finite maps from locations ( $x$ ) to values ( $v$ ). The garbage stack  $G$  is a technical device used to record the deallocated storage so that the program stays closed despite dangling pointers. Note, however, that the abstract machine becomes stuck if the program attempts to read or write a location in the garbage stack. The primary goal of the formalism is to prove that well-typed programs cannot get stuck, and thus the garbage can be safely reclaimed at any point during the execution.

### 4.1 Syntax

Figure 3 gives BNF definitions for the syntax of the statements, expressions, and types for Core Cyclone. Constructors ( $\tau$ ) define syntax for both types and regions. We use a kind discipline to determine whether a type variable represents a type ( $\mathcal{T}$ ) or a region ( $\mathcal{R}$ ).

Types include pairs  $(\tau_1 \times \tau_2)$  to model primitive structs. Like `structs`, pairs are passed by value (i.e., copy-in/copy-out). They cannot instantiate type variables because we do not duplicate polymorphic code and values of pair types are larger than other types. Types also include type variables, universal types, and existential types. The quantifiers can range over types or regions and include region constraints, which are used to specify partial orders on region lifetimes. A region constraint ( $\gamma$ ) is a list of primitive constraints of the form  $\epsilon <: \rho$  where  $\epsilon$  is a region set, and  $\rho$  is a region. Intuitively, the constraint means that if you can show any of the regions in  $\epsilon$  are live, then you can assume  $\rho$  is live. Region sets can include either region variables ( $\rho$ ) or the `regions_of` of a type variable. (We omit the `regions_of` for conciseness.) Finally, function types include a region set ( $\epsilon$ ), which specifies the function’s effect — the set of regions that must be live before calling the function.

Statements consist of expressions, return statements, composition, if-statements, and while-statements. In addition, they include blocks  $(\rho : \{\tau x_\rho = e; s\})$  for declaring a new stack region and a variable within that region, dynamic region declarations (`region( $\rho$ )  $x_\rho s$` ), and a mechanism for opening values of existential type. Finally, statements include a special form “`s pop[ $\rho$ ]`” which, when executed, evaluates  $s$  to a terminal state and then deallocates the region  $\rho$ . This special form is not available at the source level, but is used internally by the abstract machine as a marker to indicate when regions should be deallocated.

Expressions include variables  $x_\rho$  which double as locations. Each variable  $x$  lives in a given region  $\rho$ ; formally  $x_\rho$  makes this fact explicit. Expressions also include integers, functions, pointer dereference, function calls, the address-of operator, and assignment as in C. In addition, expressions include polymorphic instantiation, pairs, projection, `rnew`,

kinds	$\kappa$	$::= \mathcal{T} \mid \mathcal{R}$
type and region vars	$\alpha, \rho$	
region sets	$\epsilon$	$::= \alpha_1 \cup \dots \cup \alpha_n \cup \{\rho_1, \dots, \rho_m\}$
region constraints	$\gamma$	$::= \emptyset \mid \gamma, \epsilon <: \rho$
constructors	$\tau$	$::= \alpha \mid \text{int} \mid \tau_1 \xrightarrow{\epsilon} \tau_2 \mid \tau_1 \times \tau_2 \mid \tau * \rho \mid \text{handle}(\rho) \mid \forall \alpha: \kappa \triangleright \gamma. \tau \mid \exists \alpha: \kappa \triangleright \gamma. \tau$
expressions	$e$	$::= x_\rho \mid v \mid e(\tau) \mid (e_1, e_2) \mid e.i \mid *e \mid \text{rnew}(e_1)e_2 \mid e_1(e_2) \mid \&e \mid e_1 = e_2 \mid \text{pack}[\tau_1, e] \text{ as } \tau_2$
values	$v$	$::= i \mid f \mid \&p \mid \text{region}(\rho) \mid (v_1, v_2) \mid \text{pack}[\tau_1, v] \text{ as } \tau_2$
paths	$p$	$::= x_\rho \mid p.i$
functions	$f$	$::= \rho: (\tau_1 x_\rho) \xrightarrow{\epsilon} \tau_2 = \{s\} \mid \Lambda \alpha: \kappa \triangleright \gamma. f$
statements	$s$	$::= e \mid \text{return } e \mid s_1; s_2 \mid \text{if } (e) s_1 \text{ else } s_2 \mid \text{while } (e) s \mid \rho: \{\tau x_\rho = e; s\} \mid \text{region}(\rho) x_\rho s \mid \rho: \{\text{open}[\alpha, x_\rho] = e; s\} \mid s \text{ pop}[\rho]$

Figure 3: Abstract Syntax of Core Cyclone

existential packages, region handles, and paths. Rather than model individual memory locations, paths provide a symbolic way to refer to a component of a compound object. For instance, if the location  $x_\rho$  contains the value  $((3, 4), (5, 6))$ , then the path  $x_\rho.1$  refers to  $(3, 4)$ , and  $x_\rho.1.2$  refers to 4.

## 4.2 Static Semantics

The most important typing judgment is the one for statements. It has the form:

$$\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{\text{stmt}} s$$

Here,  $\Delta$  records the set of type and region variables that are in scope,  $\Gamma$  records the set of value variables in scope along with their types,  $\gamma$  records partial order constraints on region lifetimes from  $\Delta$ ,  $\epsilon$  records the static capability (i.e., which regions in  $\Delta$  are considered live), and  $\tau$  records the return type for the statement.

The inference rules for deriving this judgment allow dangling pointers to be manipulated, but they cannot be dereferenced. This is because the rules for pointer dereference require that the region into which the pointer refers is still live. To establish liveness, it suffices to show that the region name  $\rho$  is in  $\epsilon$ . Often, this can be shown directly. When it cannot, we can try to find another region  $\rho'$  in  $\epsilon$  such that the constraint  $\rho' :> \rho$  is in  $\gamma$ . The constraint ensures that  $\rho$  outlives  $\rho'$  and since  $\rho'$  is live,  $\rho$  must be also.

The other important judgment for statements is

$$\vdash_{\text{ret}} s$$

which asserts that the statement will not “fall off”. Rather, if execution of the statement terminates, then the terminal state will be of the form  $\text{return}(v)$  for some value  $v$ . This judgment is defined via a simple syntax-directed analysis.

Another important typing judgment is the one that allows us to assert that a garbage stack  $G$  and stack  $S$  can be described by the context  $\Delta; \Gamma; \gamma$ :

$$\vdash_{\text{heap}} (G, S) : \Delta; \Gamma; \gamma$$

Here,  $\Delta$  is the set of region names that are bound in either  $G$  or  $S$ ;  $\Gamma$  records the types of the locations bound in either  $G$  or  $S$ ; and  $\gamma$  records the relative lifetimes of the regions in  $S$ . This judgment is used to connect assumptions that a statement might make with the reality of the current heap.

With these top-level judgments (informally) defined, we can state the Soundness Theorem for Core Cyclone:

THEOREM 4.1 (SOUNDNESS). *If:*

1.  $\vdash_{\text{heap}} (\emptyset, [\rho_H \mapsto R]) : \Delta; \Gamma; \gamma$ ,
2.  $\vdash_{\text{ret}} s$ ,
3.  $\Delta; \Gamma; \gamma; \{\rho_H\}; \text{int} \vdash_{\text{stmt}} s$ , and
4.  $s$  contains no **pop** statements

then either  $(G, S, s)$  runs forever or there exists a  $G', R'$  and  $i$  such that  $(G, [\rho_H \mapsto R], s) \rightarrow^* (G', [\rho_H \mapsto R'], \text{return}(i))$

In plain English, if we start with an empty garbage heap, and a stack that contains a single heap region  $([\rho_H \mapsto R])$  that is well-formed, and if statement  $s$  doesn’t fall off, and  $s$  is well formed with respect to the type of the initial heap and promises to return only integers, and  $s$  does not contain **pop** statements, then the program cannot get stuck from type errors or dangling-pointer dereferences. Furthermore, if the program terminates, all of the regions it allocated will have been freed and the program will return an integer. The proof details are available in our companion technical report [13].

## 5. IMPLEMENTING CYCLONE REGIONS

The code-generation and run-time support for Cyclone regions is very simple. Heap and stack manipulation are exactly as in C. Dynamic regions are represented as linked lists of “pages” where each page is twice the size of the previous one. A region handle points to the beginning of the list and the current “allocation point” on the last page, where **rnew** or **rmalloc** place the next object. If there is insufficient space for an object, a new page is allocated. Region deallocation simply frees each page of the list.

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

The code generator ensures that regions are deallocated even when their lifetimes end due to unstructured control flow. For each intra-procedural jump or **return**, it is easy to determine statically how many regions should be deallocated

before transferring control. When throwing an exception, this is no longer the case. Therefore, we store region handles and exception handlers in an integrated list that operates in a last-in-first-out manner. When an exception is thrown, we traverse the list deallocating regions until we reach an exception handler. We then transfer control with `longjmp`. In this fashion, we ensure that a region is always deallocated when control returns.

## 6. EXPERIMENTAL RESULTS

One of our central goals has been to minimize the number of required region annotations, to simplify both writing new code and porting existing code. To evaluate our design, we examined a large body of Cyclone code, including applications and libraries. In this section, we present our observations, finding that region annotations impose negligible burden on the application writer, but a somewhat larger burden on the library writer.

### 6.1 Application Code

To understand the overhead of porting C code to Cyclone, and particularly the impact of our region system, we ported a number of legacy applications and compared the differences in source code between the original and the Cyclone version. We picked several networking applications because they are part of the “systems” domain in which controlling data representation is important; these include a web server (`mini_httpd`), some web utilities (`http_get`, `http_post`, `http_ping`, and `http_load`), and a simple client (`finger`). We also used some computationally-intense, older C applications that make heavy use of arrays and pointers; these include `cfrac`, `grobner`, and `tile`. Finally, we ported the compression utilities `cacm` and `ncompress`.

We took two approaches to porting. First, we changed all the programs as little as possible to make them correct Cyclone programs. Then, for `cfrac` and `mini_httpd`, we *regionized* the code: We made functions more region polymorphic and, where possible, eliminated heap allocation in favor of dynamic region allocation with `rnew`. We also added compiler-checked “not null” annotations to pointer types where possible to avoid some null checks.

The results of our efforts are summarized in Table 1. For each benchmark program, we show the number of lines of C and Cyclone code, the differences between the two, the region annotations required, and a performance comparison of the Cyclone version with or without bounds/null checks. The + column indicates the number of lines added relative to C, and the - column is the lines removed. For the annotations, the *total* column is the number of individual region-related alterations, including per-variable annotations and occurrences of `region r {s}` and `rnew`. The *lines* column is the total number of lines in the file that changed due to these annotations. Though not our focus, we give performance estimates as well. The times are the median running times (n=21) on a 750MHz PentiumIII with 256MRam running Linux kernel 2.2.16-12. The percentage for the Cyclone programs is time relative to the C version.

There are two interesting results regarding the difficulty of minimal-porting. First, the overall changes in the programs are relatively small — less than 10% of the program code needed to be changed. The vast majority of the overall differences arise from pointer-syntax alterations. These changes are typically easy to make — e.g., the type of strings

are changed from `char *` to `char ?`.

The most encouraging result is that the number of region annotations is small: only 124 changes in total for more than 18,000 lines of code, which account for roughly 6% of the total changes. The majority of these changes were completely trivial, e.g., many programs required adding  $\rho_H$  annotations to `argv` so that arguments could be stored in global variables. The program that required the most changes was `grobner`. Interestingly, the majority of these changes arose from the fact that in one place a stack pointer was being stored in a `struct` type. We therefore parameterized the `struct` definition with a region variable, and this parameterization then propagated through the rest of the code. However, the default annotation still worked in many cases: out of 133 total variable declarations of the parameterized `struct` type, only 38 required annotations, or 28%.

The cost of porting a program to use dynamic regions was also reasonable; in this case roughly 13% of the total diffs were region-related. For the web server, we were able to eliminate heap allocation entirely. Because it is event-driven, handling each request as it comes in, we changed the main handler function to create a dynamic region and then pass the region handle to its subroutines in a request structure. After the request is serviced, the region is freed. The majority of the overall changes arose from moving global variables into the request structure and adding the structure as a parameter to various functions. This request structure is parameterized by a region, so many of the functions need annotations to connect the region of the request structure to that of another argument or return value.

We were less successful in regionizing `cfrac`. As in the web server, we changed many functions to allocate using region-handle parameters. It was easy to do dynamic region allocation and deallocation as part of the algorithm’s main iteration, but for large inputs, it was difficult to keep regions from growing large before deallocation. We conclude that garbage collection is a better match for this code, but others have had more success with regions [11].

As for performance, we achieve near-zero overhead for network or I/O bound applications such as the http clients and servers, but we pay a considerable run-time penalty for processor-intensive benchmarks such as the compression tools. The unusually high overhead for the unregionized `cfrac` appears due to poor code generation for `*p++` where `p` has type `int?`. The regionized port avoids such expressions. We believe much of the overhead is due to array representation, not regions. We address this issue further in Section 8.

### 6.2 Library Code

We have ported a significant subset of the C and Ocaml libraries to Cyclone. Two illustrative cases are the Cyclone list and string libraries, ported from Ocaml and C respectively. Table 2 summarizes the region annotations in the interfaces and implementations of these libraries. As a rough measure of the effectiveness of default region annotations, we also provide results for “maximally annotated” versions of the interfaces (`list-max.h` and `string-max.h`, respectively). The *proto* column lists the number of region type annotations that were necessary in function prototypes; the *rnew* column lists the number of uses of `rnew`, and the *region* column lists the number of uses of dynamic regions.

We found that library code requires more region annotations than application code, but most of these annotations

Program	LOC		diffs		annotations		performance				
	C	Cyc	+	-	total	lines	C time (s)	checked(s)	%	unchecked(s)	%
cacm	340	359	42	23	0	0	1.77	3.49	97%	3.03	71%
cfrac	4218	4214	132	136	2	2	2.61	17.07	554%	17.07	554%
finger	158	161	18	15	3	3	0.58	0.55	-5%	0.48	-17%
grobner	3244	3377	438	305	71	40	0.07	0.20	186%	0.20	186%
http_get	529	529	36	36	4	4	0.28	0.28	0%	0.28	0%
http_load	2072	2057	115	130	15	13	89.37	90.22	1%	90.19	1%
http_ping	1072	1081	30	21	1	1	0.28	0.28	0%	0.28	0%
http_post	607	608	42	41	8	8	0.16	0.16	0%	0.16	0%
matxmult	57	48	3	12	3	1	1.38	1.83	32%	1.38	0%
mini_httpd	3005	3022	233	216	4	4	3.71	3.85	4%	3.86	4%
ncompress	1964	1982	120	102	10	9	0.20	0.39	95%	0.38	90%
tile	1345	1366	145	124	2	2	0.48	1.05	116%	0.99	104%
total	18611	18804	1354	1161	124	86	-	-	-	-	-

“regionized” versions of benchmarks

cfrac	4218	4110	501	528	158	107	2.61	10.07	286%	8.80	237%
mini_httpd	3005	2967	500	522	88	54	3.71	3.83	3%	3.82	3%
total	7223	7174	1001	1050	246	161	-	-	-	-	-

Table 1: Porting C code to Cyclone

	LOC	proto	rnew	region
string.h	139	57	0	0
string-max.h	139	135	0	0
string.cyc	739	68	14	2
list.h	364	85	0	0
list-max.h	364	171	0	0
list.cyc	819	74	38	0

Table 2: Region annotations in libraries

are for the sake of convenience and generality rather than necessity. Library functions that perform allocation tend to come in two flavors: a heap allocating function that has the same signature as the corresponding C or Ocaml function, and a region version that takes an additional region handle. Most of the annotations occur in the latter, and so were made for the sake of the convenience of using of the libraries with arbitrary regions. Most of the changes are to function prototypes; no explicit region annotations were necessary in the bodies of functions. The maximally annotated interfaces require 2-2.4 times more region annotations; that is, the default region annotations suffice 50-60% of the time. Most of the non-default region annotations were needed to express a “same-region” relationship between arguments and return types or to allow the function to allocate into an arbitrary region; the remainder were needed in type definitions. Moreover, no effect annotations whatsoever were necessary.

Most importantly, our applications, such as the compiler, use the libraries extensively and region instantiation is implicit throughout them. The vast majority of library calls in ported C code require no changes; `malloc`, `realloc`, `memcpy`, etc., are essentially the only exceptions.

## 7. RELATED WORK

In this paper, we have concentrated on the region-based type system for Cyclone, which naturally supports C-style stack allocation, conventional heap allocation, and dynamic region allocation. We feel that Cyclone is a unique and promising point in the programming-language design-space, but many other systems share some of these features.

*Making C Safe.* Many systems, including (but not limited to) LCLint [9, 8], SLAM [3], Safe-C [2], and CCured [19] aim to make C code safe. Some of these systems, such as LCLint, are meant to be static bug-finding tools. Like Cyclone, they usually require restricted coding idioms or additional annotations, but unlike Cyclone, they offer no soundness guarantees. In this way, these static tools reduce false positives. In contrast, Cyclone uses a combination of static analysis (for memory management) and run-time checks (for bounds violations) to minimize false positives.

Other systems, such as Safe-C and CCured, ensure soundness by rewriting the code and adding run-time checks. The primary advantage of these systems is that they require (almost) no changes to the C code, unlike Cyclone. However, they do not preserve the same data representations and lifetimes for objects. Furthermore, memory errors are caught at run-time instead of compile time. For instance, when an object is freed under CCured, the (entire) storage is not immediately reclaimed, but rather marked as inaccessible. Subsequent accesses check the mark and signal an error when the object is dereferenced. Ultimately, the “mark” is reclaimed with a garbage collector to avoid leaks. Furthermore, CCured may implicitly move some stack-allocated objects to the heap to avoid dangling-pointer dereferences.

*Static Regions.* Tofte and Talpin’s seminal work [22] on implementing ML with regions provides the foundation for regions in the ML Kit [21]. Programming with the Kit is convenient, as the compiler automatically infers all region annotations. However, small changes to a program can have drastic, unintuitive effects on object lifetimes. Thus, to program effectively, one must understand the analysis and try to control it indirectly by using certain idioms [21]. More recent work for the ML Kit includes optional support for garbage collection within regions [14].

A number of extensions to the basic Tofte-Talpin framework can avoid the constraints of LIFO region lifetimes. As examples, the ML Kit includes a reset-region primitive [21]; Aiken et al. provide an analysis to free some regions early [1]; and Walker et al. [23, 24] propose general systems for freeing regions based on linear types. All of these systems are

more expressive than our framework. For instance, the ideas in the Capability Calculus were used to implement type-safe garbage collectors *within* a language [25, 18]. However, these systems were not designed for source-level programming. Rather, they were designed as compiler intermediate languages or analyses and can thus ignore user issues such as minimizing annotations or providing control to the user.

Two other recent projects, Vault [7] and the work of Henglein et al. [15] aim to provide convenient, source-level control over memory management using regions. Vault’s powerful type system allows a region to be freed before it leaves scope and its types can enforce that code *must* free a region. To do so, Vault restricts region aliasing and tracks more fine-grained effects. As a result, programming in Vault requires more annotations. Nevertheless, we find Vault an extremely promising direction and hope to adapt some of these ideas to Cyclone. Henglein et al. [15] have designed a flexible region system that does not require LIFO behavior. However, the system is monomorphic and first-order; it is unclear how to extend it to support polymorphism or existential types.

**Regions in C.** Perhaps the most closely related work is Gay and Aiken’s RC [11] compiler and their earlier system, C@ [10]. As they note, region-based programming in C is an old idea; they contribute language support for efficient reference counting to detect if a region is deallocated while there remain pointers to it (that are not within it). This dynamic system has no *a priori* restrictions on regions’ lifetimes and a pointer can point anywhere, so the RC approach can encode more memory-management idioms. Like Cyclone, they provide pointer annotations. These annotations are never required, but they are often crucial for performance because they reduce the need for reference-counting. One such annotation is very similar to our notion of region subtyping.

RC uses reference counting only for dynamic regions. In fact, one annotation enforces that a pointer never points into a dynamic region, so no reference counting is needed. As a result, RC allows dangling pointers into the stack or heap. Other kinds of type errors also remain. Indeed, we found a number of array-bounds bugs in the benchmarks used to evaluate RC, such as `grobner`. Finally, RC cannot support the kind of polymorphism that Cyclone does because the RC compiler must know statically which objects are pointers.

In summary, some of these systems are more convenient to use than Cyclone (e.g., CCured and the MLKit) but take away control over memory management. Some of the static systems (e.g., the Capability Calculus) provide more powerful region constructs, but were designed as intermediate languages and do not have the programming convenience of Cyclone. Other systems (e.g., RC, Safe-C) are more flexible but offer no static guarantees.

## 8. FUTURE WORK

A great deal of work remains to achieve our goals of providing a tool to easily move legacy code to a type-safe environment and providing a type-safe language for building systems where control over data representations and memory management is an issue.

In the near future, we hope to incorporate support for deallocating dynamic regions early. We have experimented briefly with linear type systems in the style of the Capability Calculus or Vault, but have found that this approach

is generally too restrictive, especially in the context of exceptions. Instead, we are currently developing a traditional intra-procedural flow analysis to track region aliasing and region lifetimes. Again, for the inter-procedural case, we expect to add support for explicit annotations, and to use experimental evidence to drive the choice of defaults.

We also expect to incorporate better support for first-class regions, in the style of RC. The goal is to give programmers a sufficient range of options that they can use the statically checked regions most of the time, but fall back on the dynamically checked regions when needed.

In addition to enhancements to the region framework, work is needed in other areas. For instance, we have seen run-time overheads ranging from 1 to 3x for the benchmarks presented here. For compute-intensive micro-benchmarks, we have seen performance range from 1 to 10x, depending on the architecture and C compiler. We are currently working to identify the bottlenecks, but an obvious problem is that we perform all array-bounds checks at run-time. For example, when we disabled bounds checks for a matrix multiply program, overhead dropped from 30% to 0%. Fortunately, array-bounds-check elimination is a well-studied issue, and we feel confident that we can adapt a simple, but effective approach to avoid much of this cost.

The other key area we are investigating is data representation: To support dynamically sized arrays and array-bounds checks, we tag such arrays with implicit size information. Similarly, to support type-safe, discriminated unions, we add implicit tags. We are adapting ideas from DML [27] and Xanadu [26] to give programmers control over the placement of these tags. We hope this will make it easier to interface with legacy C code or devices that do not expect these tags on the data. However, we have found that the DML framework does not easily extend to imperative languages such as C. In particular, there are subtle issues involving existential types and the address-of (&) operator [12].

## Acknowledgments

We would like to thank Steve Zdancewic and Jeff Vinocur for help in proofreading this manuscript.

## 9. REFERENCES

- [1] 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, 1995.
- [2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *ACM Conference on Programming Language Design and Implementation*, pages 290–301, June 1994.
- [3] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer-Verlag, May 2001.
- [4] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.
- [5] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. *Information and Computation*, 155:108–133, 1999.

- [6] Cyclone user's manual. Technical Report 2001-1855, Department of Computer Science, Cornell University, Nov. 2001. Current version at <http://www.cs.cornell.edu/projects/cyclone/>.
- [7] 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.
- [8] D. Evans. LCLint user's guide. <http://lclint.cs.virginia.edu/guide/>.
- [9] D. Evans. Static detection of dynamic memory errors. In *ACM Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, Pennsylvania, May 1996.
- [10] D. Gay and A. Aiken. Memory management with explicit regions. In *ACM Conference on Programming Language Design and Implementation*, pages 313–323, Montreal, Canada, June 1998.
- [11] 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.
- [12] D. Grossman. Existential types for imperative languages. Oct. 2001. Submitted for publication. Available at [http://www.cs.cornell.edu/home/danieljg/papers/exists\\_imp.pdf](http://www.cs.cornell.edu/home/danieljg/papers/exists_imp.pdf).
- [13] D. Grossman, G. Morrisett, Y. Wang, T. Jim, M. Hicks, and J. Cheney. Formal type soundness for Cyclone's region system. Technical Report 2001-1856, Department of Computer Science, Cornell University, Nov. 2001. Available at [http://www.cs.cornell.edu/home/danieljg/papers/cyclone\\_regions\\_tr.pdf](http://www.cs.cornell.edu/home/danieljg/papers/cyclone_regions_tr.pdf).
- [14] N. Hallenberg. Combining garbage collection and region inference in the ML Kit. Master's thesis, Department of Computer Science, University of Copenhagen, 1999.
- [15] F. Henglein, H. Makhholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Third International Conference on Principles and Practice of Declarative Programming*, Firenze, Italy, Sept. 2001.
- [16] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, Jan. 1996.
- [17] J. Mitchell and G. Plotkin. Abstract types have existential type. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version in *Twelfth ACM Symposium on Principles of Programming Languages*, 1985.
- [18] S. Monnier, B. Saha, and Z. Shao. Principled scavenging. In *ACM Conference on Programming Language Design and Implementation*, pages 81–91, Snowbird, UT, June 2001.
- [19] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Twenty-Ninth ACM Symposium on Principles of Programming Languages*, Portland, OR, Jan. 2002. To appear.
- [20] M. Tofte and L. Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):734–767, July 1998.
- [21] M. Tofte, L. Birkedal, M. Elsmann, 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.
- [22] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
- [23] 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.
- [24] D. Walker and K. Watkins. On regions and linear types. In *ACM International Conference on Functional Programming*, pages 181–192, Sept. 2001.
- [25] D. C. Wang and A. W. Appel. Type-preserving garbage collectors. In *Twenty-Eighth ACM Symposium on Principles of Programming Languages*, pages 166–178, London, United Kingdom, Jan. 2001.
- [26] H. Xi. Imperative programming with dependent types. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science*, pages 375–387, Santa Barbara, June 2000.
- [27] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.