Region-Based Memory Management in Cyclone *

Dan Grossman Greg Morrisett Trevor Jim†
Michael Hicks Yanling Wang James Cheney

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

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

ABSTRACT

Cyclone is a type-safe programming language derived from C. The primary design goal of Cyclone is to let programmers control data representation 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 a combination of default annotations, local type inference, and a novel treatment of region effects reduces 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.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*dynamic storage management*

General Terms

Languages

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, 19], a language designed to be very close to C, but also safe. We have written or ported over 110,000 lines of Cyclone code, including the Cyclone compiler, an extensive library, lexer and parser generators, compression utilities, device drivers, a multimedia distribution overlay network, a web server, and many smaller benchmarks. In the process, we identified many common C idioms that are usually safe, but which the C type system is too weak to verify. We then augmented the language with modern features and types so that programmers can still use the 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, many uses of the addresses of local variables require no modification.

*This research was supported in part by Sloan grant BR-3734; NSF grant 9875536; AFOSR grants F49620-00-1-0198, F49620-01-1-0298, F49620-00-1-0209, and F49620-01-1-0312; ONR grant N00014-01-1-0968; and NSF Graduate Fellowships. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

PLDI'02, June 17-19, 2002, Berlin, Germany.

- *Exposed:* Programmers control where objects are allocated and how long they live. As usual, local variables are always allocated on the stack.

- *Comprehensive:* We treat all memory uniformly, including the stack, the heap (which can optionally be garbage-collected), and "growable" regions.

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

Following the seminal work of Tofte and Talpin [28], the system is *region-based*: each object lives in one region and, with the exception that a distinguished heap region may be garbage collected, a region's objects are all deallocated simultaneously. As a static system for an explicitly typed, low-level language, Cyclone's region framework makes several 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.

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

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

We have found Cyclone's region system sufficiently expressive for porting legacy C code and writing new applications. In our experience, porting C code has required altering about 8% of the code, and the vast majority of changes have not been region annotations. Furthermore, Cyclone performed as well as C for the network applications we considered, and within a factor of three for more computationally intense programs.

In this paper, we demonstrate our contributions, beginning with a general description of the system suitable for programmers (Section 2). We then present 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 burden of porting C code to Cyclone and the resulting performance (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 objects reside in the heap. Primitives `malloc` and `new` create new heap objects. The `new` operation is like `malloc` except that it takes an expression and initializes the memory with it. There is no explicit mechanism for reclaiming heap-allocated objects (e.g., `free`). However, Cyclone programs may optionally link against the Boehm-Demers-Weiser conservative garbage collector [4] to reclaim unreachable heap-allocated objects implicitly. 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 lifetime as in C.

Dynamic regions are created with the construct `region r {`s`}`, where `r` is an identifier and s is a statement. The region's lifetime is the execution of s. In s, `r` is bound to a region *handle*, which primitives `rmalloc` and `rnew` use to allocate objects into the associated region. For example, `rnew(r) 3` returns a pointer to an `int` allocated in the region of handle `r` and initialized to 3. Handles are first-class values; a caller may pass a handle to a function to allow it to allocate into the associated region. A predefined 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 τ `?`) are implemented with extra fields to support bounds-checks, but this design is orthogonal to regions. 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*`ρ describes a pointer to an `int` that is in the region whose

```
char?ρ   strcpy<ρ,ρ_2>(char?ρ d, const char?ρ_2 s);
char?ρ_H strdup<ρ>(const char?ρ s);
char?ρ   rstrdup<ρ,ρ_2>(region_t<ρ>,const char?ρ_2 s);
size_t   strlen<ρ>(const char?ρ s);
```

Figure 1: Cyclone string library prototypes

name is ρ. The invariant that pointers have a particular region is the basic restriction we impose to make the undecidable problem of detecting dangling-pointer dereferences tractable. Pointer types with different region names are different types. A handle for a region corresponding to ρ has the type `region_t`<ρ>.

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

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

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 often suffices to ensure that pointer types' region names are 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 ρ_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 `int*`ρ_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 on simple scoping mechanisms to ensure soundness. Instead, we must track the set of live region names at each control-flow point. To keep the analysis intraprocedural, we use a novel type-and-effects system to track interprocedural liveness requirements. We delay the full discussion of effects until Section 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 whether they point into 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 ρ and ρ_2 are abstracted by the syntax <ρ, ρ_2>, 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 syntax <ρ_L, ρ_H> in the call instantiates ρ_2 with the heap region ρ_H and ρ with the stack region ρ_L, allowing one to copy a string from the heap to the stack.

Region polymorphism can 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, the region name of the returned value ρ 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 region arguments. As an example, this contrived program has a function `fact` that abstracts a region ρ and takes as arguments a pointer into ρ 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 ρ_H for ρ. In contrast, the recursive call instantiates ρ with ρ_L, which is the name of the stack region. At run time, the first call to `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 these definitions to be parameterized by region names. For example, here is a declaration for lists of pointers to ints:

```
struct Lst<ρ_1,ρ_2> {
  int*ρ_1 hd;
  struct Lst<ρ_1,ρ_2> *ρ_2 tl;
};
```

Ignoring subtyping, a value of type `struct Lst`<ρ_1`,`ρ_2> is a list with `hd` fields that point into ρ_1 and `tl` fields that point into ρ_2. Other invariants are possible: If the type of `tl` were `struct Lst`<ρ_2`,`ρ_1>`*` ρ_2, the declaration would

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

Figure 2: Cyclone prototypes minimally-annotated

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<ρH,ρ> *ρ list_t<ρ>;
```

2.3 Subtyping

Although 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*ρL p;
      if(b) p = p1; else p=p2;
      /* ... do something with p ... */ }
}
```

It appears that the program should fail to type-check because neither `p1` nor `p2` has type `int*`ρ_L. If we change the type of `p` to `int*`ρ_1 or `int*`ρ_2, then one of the assignments is illegal.

To solve this problem, we observe that if the region corresponding to ρ_1 *outlives* the region corresponding to ρ_2, then it is sound to use a value of type τ`*`ρ_1 where we expect one of type τ`*`ρ_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 preconditions that express partial orders on region lifetimes. In practice, we have very rarely used this feature, because the local outlives information has sufficed.

To ensure soundness, we do not allow casting τ_1`*`ρ to τ_2`*`ρ, even if τ_1 is a subtype of τ_2, as this cast would allow putting a τ_2 in a location where other code expects a τ_1. (This problem is the usual one with covariant subtyping on references.) However, Cyclone does allow casts from τ_1`*`ρ to `const` τ_2`*`ρ_2 when τ_1 is a subtype of τ_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, we use a combination of inference and well-chosen defaults to reduce dramatically the number of annotations needed in practice. We emphasize that our approach to inference is purely intraprocedural 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 always 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 (intraprocedural) 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 generalized implicitly. 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 ρ_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. 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;
}
```

Put another way, the function above, when treated as C code, ports to Cyclone with no modification. 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<ρ1,ρ2,ρ3>(int *ρ1*ρ2 pp, int *ρ3 p) {*pp=p;}
```

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

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

Finally, we employ another technique that greatly 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<ρ1,ρ2> *ρ2 l_t<ρ1,ρ2>;
l_t heap_copy(l_t l) {
  l_t ans = NULL;
  for(l_t l2 = l; l2 != NULL; l2 = l2->tl)
    ans = new Lst(new *l2->hd,ans);
  return ans;
}
```

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

3. EFFECTS

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

But if there is some way to hide a pointer's type in a result, then the pointer could escape the scope of its region. 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 address for functional languages.

Cyclone does not have closures, but it has other typing constructs that hide regions. In particular, Cyclone provides existential types [22, 14], which suffice to encode closures [21] 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 region names that are considered live at each control-flow point. Following Walker, Crary, and Morrisett [29], we call the set of live regions the *capability*. To allow dereferencing a pointer, the type system ensures that the associated region name is in the capability. Similarly, to allow a function call, Cyclone ensures that regions the function might access are all live. To this end, function types carry an *effect* that records the set of regions the function might access. The 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 includes the set of region names that appear in the argument or result types. For instance, given the prototype:

```
int*ρ1 f(int*, int*ρ1*);
```

which elaborates to:

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

the default effect is `{`ρ_1,ρ_2,ρ_3`}`. In the absence of polymorphism, this default effect is a conservative bound on the regions the function might access. 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*ρ1 f(int*ρ2, int*ρ1*ρ3; {ρ1,ρ3});
```

In practice, we have found default effects extremely useful. Indeed, for the 110,000 lines of Cyclone code we have 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 occur free in only one location, and all effect variables had to be prenex quantified [26]. 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 needed effect variables in some library interfaces, making the libraries harder to understand and use. Consider, for instance, a type for polymorphic sets:

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

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

Suppose the library exports the `Set` structure to clients abstractly (i.e., without revealing its definition):

```
struct Set<α,ρ,ϵ>;
```

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

3.1 Avoiding Effect Variables

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

$$\texttt{regions\_of(int)} = \emptyset$$
$$\texttt{regions\_of(}\tau\texttt{*}\rho\texttt{)} = \{\rho\} \cup \texttt{regions\_of(}\tau\texttt{)}$$
$$\texttt{regions\_of(}(\tau_1,\ldots,\tau_n) \to \tau\texttt{)} =$$
$$\texttt{regions\_of(}\tau_1\texttt{)} \cup \cdots \cup \texttt{regions\_of(}\tau_n\texttt{)} \cup \texttt{regions\_of(}\tau\texttt{)}$$

For type variables, regions_of(α) is treated as an abstract set of region variables, much like effect variables. For example, regions_of(α*ρ) = {ρ} $\cup$ regions_of(α). The default effect of a function that has α in its type simply includes regions_of(α).

With the addition of regions_of, we can rewrite the Set example as follows:

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

Now the connection between the type parameter α 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(α) is the default effect for int (*cmp)(α,α), so we need not write it.

Now suppose we wish to build a Set<int*ρ_1,ρ_2> value using a particular comparison function:

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

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

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

3.2 Interaction with Existential Types

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 ∃α { int (*func)(α env); α env;};
```

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

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{<β> .func = f,.env = y} = pkg;
  return f(y);
}
```

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

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

Here, the abstracted type α is instantiated with int*ρ_L because the call-back's environment is a pointer to an int in region ρ_L. The function for the call-back just dereferences the pointer it is passed. When packaged as an existential, the int*ρ_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. Rather than resorting to effect variables, we give regions_of(α) a *bound*:

```
struct IntFn<ρ> ∃α:>ρ { ... };
```

The bound means regions_of(α) must all *outlive* ρ; the type-checker rejects an instantiation of α in which the bound may not hold. Therefore, if pkg has type struct IntFn<ρ>, then we can call f so long as ρ is live. In practice, bounds reduce the "effect" of a call-back to a single region.

4. FORMAL SOUNDNESS

In a separate technical report [15], we have defined an operational model of Core Cyclone, formalized the type system, and proven type soundness. Space constraints prevent us from including the material here, so we summarize the salient details.

Core Cyclone includes all of the features relevant to memory management, including stack allocation, dynamic regions, polymorphism, and existential types. 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 lists mapping region names (ρ) to regions (R), which in turn are maps from locations (x) to values (v). The garbage stack G is a technical device 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, so the garbage stack (the deallocated regions) need not exist during execution.

4.1 Syntax

Figure 3 gives BNF definitions for the syntax of the statements, expressions, and types for Core Cyclone. Constructors (τ) 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 structs. Like structs, pairs are passed by value (i.e., copied). We do not duplicate polymorphic code, so pair types cannot instantiate type variables because their values are larger than those of other types (i.e., they are at least two words). 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 (γ) is a list of primitive

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

Figure 3: Abstract Syntax of Core Cyclone

constraints of the form $\epsilon <: \rho$ where ϵ is a region set, and ρ is a region. Intuitively, the constraint means that if ρ is live, then any of the regions in ϵ are live. Region sets can include region variables (ρ) or the `regions_of` a type variable. (We omit the `regions_of` for conciseness.) Finally, function types include a region set (ϵ), which specifies the function's effect (i.e., 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 ($\texttt{region}\langle\rho\rangle\, x_\rho\, s$), and a form for opening values of existential type. Finally, statements include a special form "$s\,\texttt{pop}[\rho]$" that, when executed, evaluates s to a terminal state and then deallocates (moves to the garbage stack) the region ρ. This form is not available to source programs; it is used internally by the abstract machine as a marker to indicate when to deallocate a region.

Expressions include variables x_ρ, which double as locations. Each variable x lives in a given region ρ; formally x_ρ makes this fact explicit. Other expressions are integers, functions, pointer dereference, function calls, the address-of operator, and assignment as in C. In addition, expressions include type instantiation, pairs, projection, `rnew`, and existential packages. Lastly, region handles ($\texttt{region}(\rho)$) are a special form not available to source programs; creating a dynamic region with $\texttt{region}\langle\rho\rangle\, x_\rho\, s$ binds x_ρ to $\texttt{region}(\rho)$.

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_ρ 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. As in C, if p is a path, then $\&p$ is a value.

4.2 Static Semantics

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

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

Here, Δ records the type and region variables that are in scope, Γ records the value variables in scope and their types, γ records partial-order constraints relating region lifetimes, ϵ records the capability (i.e., which regions in Δ are considered live), and τ records the type that e must have in any statement of the form `return` e. We present just a few interesting rules.

Type-checking statements requires checking that expressions have the correct types. For example, the rule for return statements is:

$$\frac{\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau}{\Delta; \Gamma; \gamma; \epsilon; \tau \vdash_{\mathrm{stmt}} \texttt{return}\, e}$$

Expressions must access only memory that can be proven live from ϵ and γ. Here are two example rules:

$$\frac{\gamma \vdash \epsilon \Rightarrow \rho}{\Delta; \Gamma; \gamma; \epsilon \vdash x_\rho : \Gamma(x_\rho)} \qquad \frac{\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau * \rho \quad \gamma \vdash \epsilon \Rightarrow \rho}{\Delta; \Gamma; \gamma; \epsilon \vdash *e : \tau}$$

We use $\gamma \vdash \epsilon \Rightarrow \rho$ to prove ρ is live. Informally, we need a $\rho' \in \epsilon$ such that the partial order γ shows ρ outlives ρ'. Of course, $\rho \in \epsilon$ suffices.

We use the same idea for our subsumption rule:

$$\frac{\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau * \rho_1 \quad \gamma \vdash \rho_2 \Rightarrow \rho_1}{\Delta; \Gamma; \gamma; \epsilon \vdash e : \tau * \rho_2}$$

To type-check function calls, we use $\gamma \vdash \epsilon \Rightarrow \epsilon_1$ to mean every α and ρ in ϵ_1 can be proven live from ϵ and γ. The rule is otherwise standard:

$$\frac{\Delta; \Gamma; \gamma; \epsilon \vdash e_1 : \tau_2 \xrightarrow{\epsilon_1} \tau \quad \Delta; \Gamma; \gamma; \epsilon \vdash e_2 : \tau_2 \quad \gamma \vdash \epsilon \Rightarrow \epsilon_1}{\Delta; \Gamma; \gamma; \epsilon \vdash e_1(e_2) : \tau}$$

Here is the rule for type instantiation:

$$\frac{\Delta; \Gamma; \gamma; \epsilon \vdash e : \forall\alpha{:}\kappa \rhd \gamma_1.\tau_2 \quad \Delta \vdash \tau_1 : \kappa \quad \gamma \vdash \gamma_1[\tau_1/\alpha]}{\Delta; \Gamma; \gamma; \epsilon \vdash e\langle\tau_1\rangle : \tau_2[\tau_1/\alpha]}$$

The only novelty is ensuring that γ establishes the constraints γ_1 used when type-checking e. The judgment $\gamma \vdash \gamma'$ just means for every $\epsilon <: \rho$ in γ', we can show $\gamma \vdash \rho \Rightarrow \epsilon$. By abuse of notation, we write $\tau_2[\tau_1/\alpha]$ for the capture-avoiding substitution of τ_1 for α in τ_2 and $\gamma_1[\tau_1/\alpha]$ for the substitution of $\texttt{regions\_of}(\tau_1)$ for α in γ_1.

Another necessary judgment for statements is

$$\vdash_{\mathrm{ret}} s$$

It ensures that if execution of s terminates, then the terminal state will have the form `return` v for some value v. This judgment, defined via a simple syntax-directed analysis, enforces that functions must not "fall off" — they always return values.

To set up the proof of soundness, we define a judgment 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, Δ is the set of region names that are bound in either G or S; Γ records the types of the locations bound in either G or S; and γ records the regions' relative lifetimes. In particular, γ describes the total order 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 judgments, we can state the Soundness Theorem for Core Cyclone:

Theorem 4.1 (Soundness). *If:*

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

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

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 returns 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 soundness proof, available in our companion technical report [15], uses long and tedious progress and preservation (subject-reduction) lemmas. Here we just sketch two complications from the proof of preservation. First, our operational semantics uses type substitution, for example $(G, S, (\Lambda\alpha{:}\kappa \triangleright \gamma.f)\langle\tau\rangle) \rightarrow (G, S, f[\tau/\alpha])$. As usual, we need a substitution lemma in order to conclude the well-typedness of $f[\tau/\alpha]$ given the well-typedness of $\Lambda\alpha{:}\kappa \triangleright \gamma.f$. Because of explicit effects and partial orders, proving the necessary substitution lemma requires several auxiliary lemmas, for example $\gamma \vdash \epsilon_1 \Rightarrow \epsilon_2$ implies $\gamma[\epsilon_3/\alpha] \vdash \epsilon_1[\epsilon_3/\alpha] \Rightarrow \epsilon_2[\epsilon_3/\alpha]$.

Second, we must weaken the theorem's assumptions that the heap has one region and s has no `pop` statements, while still proving that the program properly deallocates all the regions it allocates. To do so, we assume that given (G, S, s), we can partition S into $S_1 S_2$ such that s deallocates all regions in S_2 (in last-in-first-out order) and none of the regions in S_1. (To see this assumption is a proper weakening, let $S_1 = [\rho_H \mapsto R]$ and $S_2 = \emptyset$.) This assumption (formalized as another judgment on statements) implies enough about the position of `pop` statements in s to prove that the program s' resulting from a rewriting step properly deallocates exactly all of the live regions not in S_1. In other words, the ability to partition S such that the necessary properties hold is preserved under evaluation.

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 all (live) dynamic regions must be scanned, and no objects within (or reachable from) dynamic regions are reclaimed.

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

6. EXPERIMENTAL RESULTS

To simplify porting to and programming in Cyclone, we have sought to minimize the number of required region annotations. Just as important, we have sought to achieve good performance. In Sections 6.1 and 6.2, we analyze the burden of porting, in terms of added annotations, and find that annotations impose negligible burden on the application writer, but a somewhat larger burden on the library writer. In Section 6.3, we present a comparison of Cyclone's performance to that of C for our ported applications, and find that while networking programs essentially perform the same as C, compute-bound applications are up to a factor of three slower due to run-time checks and pointer representations.

6.1 Porting Application Code

We ported a number of 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

Program	LOC			annotations	
	C	Cyc	diffs	total	lines
cacm	340	360	41	0	0
cfrac	4218	4215	134	2	2
finger	158	161	17	3	3
grobner	3260	3401	452	71	40
http_get	529	530	44	4	4
http_load	2072	2058	121	15	13
http_ping	1072	1082	33	1	1
http_post	607	609	51	8	8
matxmult	57	53	11	3	1
mini_httpd	3005	3027	266	4	4
ncompress	1964	1986	134	10	9
tile	1345	1365	148	2	2
total	18627	18847	1452	124	86
regionized benchmarks					
cfrac	4218	4192	503	158	107
mini_httpd	3005	2986	531	88	54
total	7223	7178	1034	246	161

Table 1: Benchmark code differences

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.

Our results are summarized in Table 1. For each program, Table 1 shows the number of lines of C and Cyclone code, the number of differences between the two, and the region annotations required in Cyclone. The *diffs* column indicates the number of lines added or changed in porting from C to Cyclone. 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.

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 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 ?`. We are currently experimenting with interpreting `char *` as a safe null-terminated string type by default; doing so allows many fewer changes.

The most encouraging result is that the number of region annotations is small: only 124 changes (which account for roughly 6% of the total changes) in more than 18,000 lines of code. The majority of these changes were completely trivial, e.g., many programs required adding ρ_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.

The cost of porting a program to use dynamic regions was also reasonable; in this case roughly 13% of the total differences 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 [12].

	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

6.2 Porting Library Code

We have ported a significant subset of the C and Caml libraries to Cyclone. Two illustrative cases are the Cyclone list and string libraries, ported from Caml 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 are for the sake of convenience and generality rather than necessity. Library functions that perform allocation often come in two flavors: a heap allocating function that has the same signature as the corresponding C or Caml function, and a version that takes an additional region handle for generality; most annotations occur in the latter. 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 im-

Test	C time(s)	Cyclone time			
		checked(s)	factor	unchecked(s)	factor
cacm	0.12 ± 0.00	0.15 ± 0.00	1.25×	0.14 ± 0.00	1.17×
cfrac†	2.30 ± 0.00	5.57 ± 0.01	2.42×	4.77 ± 0.01	2.07×
finger	0.54 ± 0.42	0.48 ± 0.15	0.89×	0.53 ± 0.16	0.98×
grobner†	0.03 ± 0.00	0.07 ± 0.00	2.85×	0.07 ± 0.00	2.49×
http_get	0.32 ± 0.03	0.33 ± 0.02	1.03×	0.32 ± 0.06	1.00×
http_load†	0.16 ± 0.00	0.16 ± 0.00	1.00×	0.16 ± 0.00	1.00×
http_ping	0.06 ± 0.02	0.06 ± 0.02	1.00×	0.06 ± 0.01	1.00×
http_post	0.04 ± 0.01	0.04 ± 0.00	1.00×	0.04 ± 0.01	1.00×
matxmult	1.37 ± 0.00	1.50 ± 0.00	1.09×	1.37 ± 0.00	1.00×
mini_httpd-1.15c	2.05 ± 0.00	2.09 ± 0.00	1.02×	2.09 ± 0.00	1.02×
ncompress-4.2.4	0.14 ± 0.01	0.19 ± 0.00	1.36×	0.18 ± 0.00	1.29×
tile†	0.44 ± 0.00	0.74 ± 0.00	1.68×	0.67 ± 0.00	1.52×

†Compiled with the garbage collector

regionized benchmarks

cfrac	2.30 ± 0.00	5.22 ± 0.01	2.27×	4.56 ± 0.01	1.98×
mini_httpd	2.30 ± 0.00	2.35 ± 0.00	1.02×	2.35 ± 0.00	1.02×

Table 3: Benchmark performance

plicit throughout them. The vast majority of library calls in ported C code require no changes; `malloc`, `realloc`, `memcpy`, etc., are essentially the only exceptions.

6.3 Performance

Table 3 shows the performance of the original C versions of our benchmark programs together with the Cyclone versions with or without bounds-checks and null-checks. We ran each benchmark twenty-one times on a 750 MHz Pentium III with 256MB of RAM, running Linux kernel 2.2.16-12, using `gcc` 2.96 as a back end. The `gcc` optimization flags used for compiling both the original C code and the output of the Cyclone compiler were `-O3 -march=i686`. Because we observed skewed distributions for the http benchmarks, we report medians and semi-interquartile ranges (SIQR).[1] For the non-web benchmarks (and some of the web benchmarks) the median and mean were essentially identical, and the standard deviation was at most 2% of the mean. The *factor* columns for the Cyclone programs show the slowdown factor relative to the C versions.

We achieve near-zero overhead for network or I/O bound applications such as the http clients and servers, but we pay a substantial penalty for compute-intensive benchmarks; the worst is `grobner`, which is almost a factor of three slower than the C version. We have seen slowdowns of a factor of six in pathological scenarios involving pointer arithmetic in some microbenchmarks.

Two common sources of overhead in safe languages are garbage collection and bounds checking. Garbage-collection overhead is not easy to measure in Cyclone, because regionizing a program can require significant work. As shown in Table 3, only a few of our benchmarks needed garbage collection. Profiling the garbage collected version of `cfrac` suggests that garbage collection accounts for approximately half of its overhead. Partially regionizing `cfrac` resulted in an 6% improvement. On the other hand, `http_load` and `tile` make relatively little use of dynamic allocation, so they have almost no garbage-collection overhead. Therefore, we expect that the overhead will vary widely for different programs depending on their memory-usage patterns.

[1] The semi-interquartile range is the difference between the high quartile and the low quartile divided by 2. This is a measure of variability, similar to standard deviation, recommended by Jain [18] for skewed distributions.

As Table 3 demonstrates, bounds-checks are also an important component of the overhead, but less than we expected. We found that a major cost is due to the representation of fat pointers. A fat pointer is represented with three words: the base address, the bounds address, and the current pointer location (essentially the same representation used by McGary's bounded pointers [20]). The result is a larger space overhead, larger cache footprint, more parameter passing and return-value copying, and increased register pressure, especially on the register-impoverished x86.

Because fat pointers are currently the only pointer types in Cyclone that support pointer arithmetic and dynamically sized arrays, good fat-pointer performance is crucial to many Cyclone programs. We found that slight changes to fat pointer operations and `gcc` flags relating to instruction selection could have a huge impact on performance. In particular, replacing inlined pointer operations with macros and setting the architecture-specific instruction-selection flag properly doubled the speed of some applications.

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 features with Cyclone.

Making C Safe. Many systems, including but certainly not limited to LCLint [10, 9], SLAM [3], Safe-C [2], and CCured [25], 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 a static type system (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, at least whenever an implementation-dependent static analysis cannot eliminate the 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. (Cyclone's τ`?` pointers also use a wide representation, but the use of these pointers is under programmer control.) 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. Moreover, CCured may move some stack-allocated objects to the heap to avoid dangling-pointer dereferences.

Static Regions. Tofte and Talpin's seminal work [28] on implementing ML with regions provides the foundation for regions in the ML Kit [27]. 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 [27]. More recent work for the ML Kit includes optional support for garbage collection within regions [16].

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 [27]; Aiken et al. provide an analysis to free some regions early [1]; and Walker et al. [29, 30] 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 [31, 23]. However, these systems were not designed for source-level programming. They were designed as compiler intermediate languages or analyses, so they can ignore issues such as minimizing annotations or providing control to the user.

Two other recent projects, Vault [7] and the work of Henglein et al. [17] aim to provide safe source-level control over memory management using regions. Vault's powerful type system allows a region to be freed before it leaves scope and its types can enforce that code *must* free a region. To do so, Vault restricts region aliasing and tracks more fine-grained effects. As a result, programming in Vault requires more annotations. Nevertheless, we find Vault an extremely promising direction and hope to adapt some of these ideas to Cyclone. Henglein et al. [17] 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.

Finally, both TAL [24] and the Microsoft CIL [13] provide some support for type-safe stack allocation. But neither system allows programmers to mix stack and heap pointers, and both systems place overly strong restrictions on how stack pointers can be used. For instance, the Microsoft CIL prevents such pointers from being placed in data structures or returned as results — features that language implementors need for effective compilation [8].

Regions in C. Perhaps the most closely related work is Gay and Aiken's RC [12] compiler and their earlier system, C@ [11]. 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 two of the benchmarks used to evaluate RC: `grobner` and `tile`. 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 move legacy code to a type-safe environment easily 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 intraprocedural flow analysis to track region aliasing and region lifetimes. Again, for the interprocedural 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 system, work is needed in other areas. For instance, we have seen run-time overheads ranging from 1x to 3x for the benchmarks presented here, and overheads as high as 6x for some compute-intensive microbenchmarks. We are currently working to identify the bottlenecks, but a clear problem is with our representation of pointers to dynamically sized arrays (`?` pointers). To support dynamically sized arrays and 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 [33] and Xanadu [32] to make these tags explicit so that programmers can control where these tags are placed. We hope doing so will make it easier to interface with legacy C code or devices that do not expect these tags on the data, and to support time-saving and space-saving optimizations. However, we have found that the DML framework does not easily extend to imperative languages such as Cyclone. In particular, there are subtle issues involving existential types and the address-of (&) operator [14].

Acknowledgments

We would like to thank David Walker for fruitful discussions, and Steve Zdancewic and Jeff Vinocur for 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, Orlando, FL, 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, Toronto, Canada, May 2001. Springer-Verlag.

[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] T. Dowd, F. Henderson, and P. Ross. Compiling Mercury to the .NET common language runtime. In N. Benton and A. Kennedy, editors, *BABEL'01: First International Workshop on Multi-Language Infrastructure and Interoperability*, volume 59.1 of *Electronic Notes in Theoretical Computer Science*, Florence, Italy, Sept. 2001.

[9] D. Evans. LCLint user's guide. `http://lclint.cs.virginia.edu/guide/`.

[10] D. Evans. Static detection of dynamic memory errors. In *ACM Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, PA, May 1996.

[11] 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.

[12] 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.

[13] A. D. Gordon and D. Syme. Typing a multi-language intermediate code. In *Twenty-Eighth ACM Symposium on Principles of Programming Languages*, pages 248–260, London, United Kingdom, Jan. 2001.

[14] D. Grossman. Existential types for imperative languages. In *Eleventh European Symposium on Programming*, pages 21–35, Grenoble, France, Apr. 2002.

[15] 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.

[16] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *ACM Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002. This volume.

[17] F. Henglein, H. Makholm, 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*, Florence, Italy, Sept. 2001.

[18] R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley, 1991.

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

[20] G. McGary. Bounds checking projects. `http://www.gnu.org/software/gcc/projects/bp/main.html`.

[21] 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, FL, Jan. 1996.

[22] J. Mitchell and G. Plotkin. Abstract types have existential type. *ACM Transactions on Progamming Languages and Systems*, 10(3):470–502, 1988. Preliminary version in Twelfth ACM Symposium on Principles of Programming Languages, 1985.

[23] 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.

[24] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In *Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 28–52, Kyoto, Japan, Mar. 1998. Springer-Verlag.

[25] 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*, pages 128–139, Portland, OR, Jan. 2002.

[26] M. Tofte and L. Birkedal. A region inference algorithm. *ACM Transactions on Progamming Languages and Systems*, 20(4):734–767, July 1998.

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

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

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

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

[31] 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.

[32] H. Xi. Imperative programming with dependent types. In *Fifteenth IEEE Symposium on Logic in Computer Science*, pages 375–387, Santa Barbara, CA, June 2000.

[33] H. Xi and F. Pfenning. Dependent types in practical programming. In *Twenty-Sixth ACM Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, TX, Jan. 1999.