Region-Based Memory Management in Cyclone

Dan Grossman Greg Morrisett Trevor Jim[†]
Mike 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 polymorphic, type-safe programming language derived from C. The primary design goals of Cyclone are to give programmers control over low-level 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 new advancements, including support for region subtyping and a coherent integration with stack allocation and a garbage collector. Cyclone requires programmers to provide some explicit region annotations to support these features, but uses a combination of default annotations, local type inference, and a novel treatment of region effects to reduce the necessary annotations. As a result, we achieve separate compilation and a relatively smooth integration of C idioms in a region-based framework.

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. But in providing this low-level power, C admits a wide class of dangerous — and extremely common — safety violations, including incorrect type casts, array-bounds violations, 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 the programmer the control needed to realize low-level systems. Moreover, porting or extending legacy code is often prohibitively expensive. Therefore, having a safe language at the C level of abstraction, with a relatively easy porting path, is an attractive alternative.

Toward this end, we have developed *Cyclone* [6], a language 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 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 framework for preventing dangling-pointer dereferences and space leaks (i.e., memory management). The design addresses several seemingly conflicting goals. Specifically, the system is:

- *Sound:* Programs never dereference dangling pointers.
- *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:* Memory-management is under programmer control. We do not violate C programmers' assumptions about where objects are allocated, how much space they consume, or how they are accessed. Specifically, local declarations are always stack-allocated.
- *Mostly Static:* Dereferencing a dangling pointer is a compile-time error. There are no run-time checks to determine if memory has been deallocated, but the system can optionally integrate a garbage-collected heap.
- *Comprehensive:* We treat all memory — including the stack, the heap, and "growable" regions — uniformly.
- *Scalable:* The system supports separate compilation; in fact, all analysis is intra-procedural.

Following the seminal work of Tofte and Talpin [21], the system is *region-based*: each object lives in one region and, with the optional exception of the heap, 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 contributions over previous work:

- A smooth integration of pointers into the stack, the garbage-collected heap, and "growable" regions: The type system has enough polymorphism to allow functions that operate over all such pointers.

- Minimal annotations: A combination of local type inference, default annotations, parameterized type definitions, and a novel effect system conspire to make the number of required annotations palatable.

- Pointer subtyping based on regions: A last-in-first-out discipline on region lifetimes induces an "outlives" relationship on regions, which, in turn, induces a useful subtyping discipline on pointer types.

- A simple run-time system: Cyclone soundly integrates region manipulation, garbage collection, and exceptions without unnecessary overhead.

- Integration of explicit regions with function pointers and first-class universal and existential types.

In the rest of this paper, we demonstrate these contributions through a description of the system suitable for C programmers (Section 2), a core formal language that we have proven sound (Section 4), an overview of our implementation (Section 5), and a preliminary study of the cost of porting C code to Cyclone (Section 6). We discuss related work in Section 7 and conclude 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. Finally, we show how regions interact with polymorphic functions, type constructors, and existential types. 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 associated dynamic region 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 τ `?`) 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 Cyclone pointer types have a *region name*, which is a compile-time identifier corresponding to a run-time region. Ignoring subtyping, `int*`ρ describes a pointer to an `int` that is in the region described by ρ. Pointer types with different region names are different types.

The region name for the heap is ρ_H. A block labeled `L` (such as `L:{int x=0;` s`}`) uses ρ_L as the region-name for the stack region that the block creates. For the region that `region r {`s`}` creates, the region name is ρ_r. The scope of region names corresponds to the lifetimes of the corresponding regions. A handle for a region corresponding to ρ has the type `region_t<`ρ`>`. That is, `r` has type `region_t<`ρ_r`>`.

We can now give types to some of the relevant constructs in Cyclone. For example, 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.

To dereference a pointer, safety demands that its region is live. Because pointer types' region names must be in scope, it appears that no well-typed pointer could be a dangling reference. For example, consider the following buggy code:

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

When executed, the code creates storage for `x` at line 2 and deallocates it at line 4. Therefore, 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 use some other 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 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 scalable (intra-procedural) and to deal with function pointers, we use a novel type-and-effects system to track inter-procedural liveness requirements. We delay the full discussion of effects until Section 2.5.

Functions. Cyclone supports region-polymorphism, so programmers can write functions that abstract the actual regions for arguments or results. 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. This allows us to pass a pointer to the locally declared variable `x`. At run time, the first instance of `fact` modifies `g`; each recursive call modifies the value of `x` in its caller's stack frame. Alternatively, we could have written the function as:

```
void fact2<ρ>(int*ρ result, int n) {
  if(n > 1) fact2<ρ>(result,n-1);
     *result *= n;
}
```

Here is yet another version that uses a dynamic region to hold all of the intermediate results:

```
void fact3<ρ>(region_t<ρ> h,int*ρ result,int n) {
  int*ρ x = rnew(h)(1);
  if (n > 1) fact<ρ>(h,x,n-1);
  *result = (*x)*n;
}
```

```
int main() {
  region r {
    int*ρ_r g = rnew(r) 0;
    return fact3<ρ_r>(r, g, 6);
  }
}
```

The function `main` creates a dynamic region ρ_r with handle `r`. It then allocates an initial result pointer `g` within ρ_r using the expression `rnew(r) 0`. Next, it calls `fact3`, instantiating ρ with ρ_r, and passing in the handle for ρ_r, the initial result pointer, and 6. At a recursive call, instead of locally stack-allocating its own result, `fact3` uses the handle it is given to allocate the pointer within ρ_r. As a result, the intermediate results will not be reclaimed until control returns to `main`. This contrived example demonstrates that dynamic regions, such as ρ_r above, can be dynamically grown by a callee, even though the region is defined within a previous caller.

By using the same region name, function prototypes can assume and guarantee region equalities of unknown regions. In the examples below, `f1` does not type-check because it might assign a pointer into the wrong region, but `f2` does type-check because the left- and right-hand-sides of the assignment have the same type:

```
void f1<ρ_1,ρ_2>(int*ρ_1*ρ_2 pp, int*ρ_3 p) { *pp = p; }
void f2<ρ_1,ρ_2>(int*ρ_1*ρ_2 pp, int*ρ_1 p) { *pp = p; }
```

Region equalities are crucial for return types, particularly when the return value is placed in a caller-specified region:

```
int*ρ identity<ρ>(int*ρ p) {return p;}
int*ρ newzero<ρ>(region_t<ρ> h) {return rnew(h) 0;}
```

The term `newzero<`ρ_H`>(heap_region)` has type `int*`ρ_H because `heap_region` has type `region_t<`ρ_H`>`.

Type Definitions. Because `struct` definitions can contain pointers, Cyclone allows `struct`s to be parameterized by region names. For example, here is a type 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`>` will be 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, we would describe lists where the regions for `hd` and `tl` alternated at each element.

Type abbreviations using `typedef` can also have region parameters. As an example, `typedef struct Lst<`ρ_H`,`ρ`> list_t<`ρ`>;` defines region-allocated lists of heap-allocated pointers.

2.3 Subtyping

If the region corresponding to ρ_1 *outlives* the region corresponding to ρ_2, then it is sound to cast from type τ`*`ρ_1 to type τ`*`ρ_2. Cyclone supports such casts implicitly. The last-in-first-out region discipline makes such outlives relationships common: when we create a region, we know every region currently live will outlive it. In practice, we have found this subtype polymorphism crucial for type-checking common C idioms. For example, consider this function:

```
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
     }
}
```

Without subtyping, the program fails to type-check because neither p1 nor p2 has type int*ρ_L. We cannot change the type of p to int*ρ_1 or int*ρ_2, for then one of the assignments would fail.

To allow functions to assume outlives relationships among unknown regions, Cyclone supports explicit pre-conditions for expressing 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 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 Convenient Defaults

For clarity, the discussion so far has used completely explicit types. In practice, fewer annotations are necessary due to a combination of inference and defaults described below. 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 maintains separate compilation.

When a code block does not have a label, the type-checker generates a fresh region name for it. This works well because we rarely need to explicitly mention region names.

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

Similarly, when calling a (region) polymorphic function, the programmer can omit the explicit region-name instantiation and the inference engine discovers it.

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. As a result, we can rewrite our fact example as:

```
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. Similarly, we can elide all region annotations in fact2.

If we had written just a prototype for fact:

```
void fact(int * result, int n);
```

then the type-checker would insert a fresh region name (ρ) for the pointer type, and generalize it, thereby elaborating the declaration to:

```
void fact<ρ>(int *ρ result, int n);
```

The point is that the prototype of the function is determined independently of the function body.

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. For example, if we try to write:

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

then the code will elaborate 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; }
```

In short, region annotations are reduced to expressing equalities necessary for safety.

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. But it does not work well when functions return one of their parameters. On the other hand, without looking at the body of the function, we cannot determine which parameter (or component of a parameter) the function might return.

Finally, 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 assigns ans the type l_t<ρ_H,ρ_H> and l2 the type l_t<ρ_1,ρ_2>.

2.5 Effects

We argued above 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 τ*ρ outside the scope of ρ's definition, simply because you cannot write down a (well-formed) type for the result[1]. Indeed, if Cyclone had no mechanisms for type abstraction, this would be the case.

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

[1]Note that the implementation must use an extended occurs check to ensure that region variables do not escape their scope as part of inference.

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 addressed in the context of a functional language with closures.

Cyclone does not have closures, but it has other typing mechanisms for hiding regions. In particular, Cyclone provides support for existential types [16], which suffice to encode closures [15] or simple forms of objects [5], among other things. 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 [22], 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 our 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 originally due to Tofte and Talpin (hereafter TT). However, our treatment of effects differs substantially from that of TT and other related 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. This is necessary to preserve our goal of separate compilation. The default effect for a function is simply the set of regions that appear in its 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 $\{\rho_1, \rho_2, \rho_3\}$ because these are the regions that show up in the argument and result types. In the absence of polymorphism, this default effect is a conservative upper bound on the regions the function might access. As with default regions in prototypes, the programmer can override the default with an explicit effect that mentions only the regions that the function actually uses. For example, if `f` never dereferences its first argument in region ρ_2, 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 that the default effect works extremely well. Indeed, for the 70,000 lines of Cyclone code we have written thus far, we have used a non-default effect for only one function.

The second major departure from TT is that we do not have *effect variables*. Effect variables were 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. These are all important issues and without some typing support, the region framework would be severely handicapped.

Therefore, in our original Cyclone design, we tried to use TT-style effect variables. However, we quickly 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 [19]. Their unification algorithm depended crucially upon these structural invariants. In an explicitly-typed language, we found that enforcing these constraints was difficult, and robbed us of expressivness. For instance, prenex quantification restriction prevented first-class polymorphic functions, which Cyclone supports.

Second, we found that effect variables appeared in the interfaces of libraries and that they made the libraries harder to understand and use — especially for C programmers. Consider, for instance, a simple implementation of 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 paired with the list so that we can determine membership in the set. 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.

Now the library might export 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.

To simplify the system and at the same time solve the expressiveness problems that effect variables were meant to address, we replaced them with a type-level operator `regions_of(`τ`)`. Intuitively, `regions_of(`τ`)` returns the set of regions that occur free in τ. In particular:

$$\texttt{regions\_of(int)} = \emptyset$$
$$\texttt{regions\_of(}\tau\texttt{*}\rho\texttt{)} = \{\rho\} \cup \texttt{regions\_of(}\tau\texttt{)}$$

In general, for a type constructor C<$\tau_1, \ldots, \tau_n$> we have:

$$\texttt{regions\_of(}C\texttt{<}\tau_1,\ldots,\tau_n\texttt{>)} = \texttt{regions\_of(}\tau_1\texttt{)} \cup \cdots \cup \texttt{regions\_of(}\tau_n\texttt{)}$$

When `regions_of` is applied to a type variable α, it is treated as an abstract set of region variables, in the same fashion as effect variables. For example, `regions_of(`α`*`ρ`)` $= \{\rho\} \cup$ `regions_of(`α`)`, which is irreducible.

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. This simplifies the interface for clients considerably.

Introducing `regions_of` had other pleasant effects[2]. For instance, in the absence of polymorphism, the default effect for a function with type $(\tau_1, \ldots, \tau_n) \rightarrow \tau$ is simply `regions_of(`$(\tau_1, \ldots, \tau_n) \rightarrow \tau$`)`. Therefore, the `Set` example can be simplified even further so as to use the default effect:

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

Now suppose we wish to build a `Set<int*ρ1,ρ2>` value using some pre-defined 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>};
}
```

Almost always, the comparison function will be defined using the default effect, which in the case of `cmp_ptr` is $\{\rho_1\}$. When we instantiate α with `int*`ρ_1, then the effect of `cmp`, becomes `regions_of(int*`ρ_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, we have found that this approach avoids the need for programmers to explicitly mention effects when designing or using libraries.

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

Of course, if programmers use stronger effects, then unification can fail. Consider the following comparison function:

```
int cmp_ptr2<ρ1>(int*ρ1 p1, int*ρ1 p2; ∅) {
  p1 == p2;
}
```

Because the code does not dereference the pointers, it does not need ρ_1 in its effect. If we try to build a `Set` using this comparison function, the type-checker will rejects the code, because $\emptyset$ is not the default effect. The programmer then has two choices: change `cmp_ptr2` to use the default effect, or rewrite the set abstraction to allow a more refined effect on the comparison function. The latter option can always be accomplished by using an extra type variable and explicit effect:

```
struct Set2<α,ρ,β> {
  list_t<α,ρ> elts;
  int (*cmp)(α,α;regions_of(β));
}
```

In this case, we can use `Set2` with the `cmp_ptr2` function, as long as we instantiate β with some type τ such that `regions_of(`τ`)` $= \emptyset$. In other words, it is always possible to recover the expressiveness of effect variables by introducing slack type variables. However, in practice, we have never had to do this.

[2]No pun intended.

3. INTERACTION WITH EXISTENTIALS

As mentioned earlier, Cyclone supports *existential types*, which allow programmers to encode closures and simple objects. 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, meant to 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 correctly initialized.

To access the fields of existential objects, we must "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, in spite of the fact that β is in scope. Indeed, the regions may not be live as demonstrated by the following function:

```
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 environment of the callback is a pointer to an integer variable `x` in region ρ_L. The function for the callback, when given the pointer, simply dereferences it. When packaged as an existential, the `int*`ρ_L is hidden and thus the result is well-typed in spite of 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. The way we accomplish this is to give `regions_of(`α`)` a *bound*:

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

The bound means `regions_of(`α`)` must all *outlive* ρ. Therefore, if `pkg` has type `struct IntFn<ρ>`, then we can call the `func` field so long as ρ is live. In practice, this soundly reduces the "effect" of the callback to a single region.

4. FORMAL SOUNDNESS

In a separate technical report [?], we have defined an operational model of core Cyclone, formalized the type system, and proven a type soundness result. 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 specified as 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 (ρ) 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 as to keep the program closed in spite of dangling pointers. Note, however, that the abstract machine becomes stuck if the program attemps 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.

Figure 1 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 primitive structs. Like `structs`, pairs are not boxed types and are passed by value (i.e., copy-in/copy-out). They cannot instantiate type variables because we do not duplicate polymorphic code and pair 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 (γ) is a list of primitive constraints of the form $\epsilon <: \rho$ where ϵ is a region set, and ρ is a region. Intuitively, the constraint means that if you can show any of the regions in ϵ are live, then you can assume ρ is live. Region sets can include either 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 — 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 mechanism for opening values of existential type. Finally, statements include a special form "$s\,\texttt{pop}[\rho]$" which, when executed, evaluates s to a terminal state and then deallocates the region ρ. 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_ρ which double as locations. Each variable x lives in a given region ρ; formally x_ρ 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`, 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_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. Expressions include one special form, $\texttt{call}\{s\}$ which, like the pop-form, is not allowed in source programs and is only used during evaluation at the point where an expression performs a function call. The expression $\texttt{call}\{s\}$ evaluates s to a terminal state of the form $\texttt{return}\, v$ and then becomes the value v.

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

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

Here, Δ records the set of type and region variables that are in scope, Γ records the set of value variables in scope along with their types, γ records partial order constraints on region lifetimes from Δ, ϵ records the static capability (i.e., which regions in Δ are considered live), and τ 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 ρ is in ϵ. Often, this can be shown directly. When it cannot, we can try to find another region ρ' in ϵ such that the constraint $\rho' :> \rho$ is in γ. The constraint ensures that ρ outlives ρ' and since ρ' is live, ρ must be also.

There is one more important judgment for statements:

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

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 $\texttt{return}(v)$ for some value v. This judgment is defined as a straightforward, 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 $\Delta; \Gamma; \gamma$:

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

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 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 machine state.

With these top-level judgments (informally) defined, 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. $\Delta; \Gamma; \gamma; \{\rho_H\}; \mathrm{int} \vdash_{stmt} s$, *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 promises to return only integers, and s does not contain `pop` statements, 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 before termination and the program will return an integer value. The proof of this theorem follows from straightforward, though long and tedious Progress and Preservation

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

Figure 1: Syntax of Core Cyclone

lemmas. All of these details may be found in our companion technical report [?].

5. IMPLEMENTING CYCLONE REGIONS

The code-generation and run-time support for Cyclone regions is very simple. Heap and stack manipulation are exactly like in C. Dynamic regions are represented as linked lists of "pages" where each page is double 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 the object, a new page is allocated and linked in. 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 an explicit deallocation, which we use for region deallocation. 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 extreme simplicity. The disadvantage is that at collection time, every object in every dynamic region appears reachable, and thus no objects within dynamic regions (or the stack) are reclaimed.

Region handles and exception handlers are kept in an integrated list that operates in a last-in-first-out manner.

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

When throwing an exception, we do not know statically how many regions should be deallocated, but the global list tells us: 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 eventually deallocated, as long as control returns.

6. EXPERIMENTAL RESULTS

One of our central goals has been to minimize the number of required region annotations on Cyclone code, to simplify both writing new code and porting existing code. To evaluate the effectiveness of our design, we examined a large body of Cyclone code, considering both 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-based memory management 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 chose 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 get them to compile correctly. Then, for `cfrac` and `mini_httpd`, we *regionized* programs; this included making them region polymorphic through the use of region annotations, and eliminating heap allocation where possible in favor of dynamic region allocation with `rnew`. We also changed `*` pointers to `@` pointers, where possible, to eliminate unnecessary 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, and the region annotations required. The diffs were generated automatically using the `diff` tool, where the `+` column indicates all of lines added to the Cyclone program from C program, while the `-` column are those lines removed. For the annotations, the *tot* 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. The top portion of the table are the "quick port" results, while the bottom part shows the regionized program results.

For the minimally-altered programs, there are two interesting results. First, the overall changes in the programs are relatively small—less than 10% of the program code needed

Program	LOC		diffs		annots	
	C	Cyc	+	-	tot	lines
cacm	340	359	42	23	0	0
cfrac	4218	4205	139	152	2	2
finger	158	161	20	19	3	3
grobner	3244	3377	462	355	71	40
http_get	529	529	37	43	4	4
http_load	1590	1579	96	113	15	13
http_ping	1042	1051	32	26	1	1
http_post	607	608	43	53	8	8
mini_httpd	3005	3014	238	229	4	4
ncompress	1964	1982	125	117	10	9
tile	1345	1366	164	143	2	2
total	17974	18179	1398	1273	124	86
"regionized" versions of benchmarks						
cfrac	4218	4110	568	676	158	107
mini_httpd	3005	2967	514	552	88	54
total	7223	7077	1082	1228	246	161

Table 1: Porting C code to Cyclone

to be changed. The vast majority of the overall differences arise from pointer syntax alterations. These changes are typically easy to make—*e.g.* all `char *` occurrences are changed to `char ?` and occurrences of pointers expected to be non-null are changed from `*` to `@`.

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 comprise roughly 6% of the total changes. The majority of these changes were completely trivial, *e.g.* many programs required adding `'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 had to bubble through the rest of the code. However, the default annotation still worked in many cases: out of 133 total variable declarations of the parameterized structure type, only 38 required annotations, or 28%.

The cost of porting a program that takes advantage of dynamic regions was also reasonable; in this case roughly 13% of the total diffs were region-related. For the webserver, we were able to eliminate heap allocation entirely. Because it is event-driven, handling each request as it comes in, we could change the main handler function to create a dynamic region and then to pass the region handle to its subroutines in a request structure. After the request was serviced, the region was 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 region, and so many of the subroutines needed annotations to tie the region of the request structure to that of some other argument or the value being returned.

XXX Dan—say something about cfrac here.

6.2 Library Code

James text + table here

7. RELATED WORK

In this paper, we have concentrated on the region-based type system for Cyclone. The type-system uses a combination of intra-procedural inference, simple effects, and default annotations to achieve a flexible, convenient framework for type-safe memory management. In particular, Cyclone naturally supports C-style stack allocation, conventional heap allocation, as well as Tofte-Talpin-style 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. We describe some of these systems, roughly organized by feature.

Making C Safe. A number of systems, including (but certainly not limited to) LCLint [9, 8], SLAM [3], Safe-C [2], and CCured [18] are aimed at making legacy C code safe. Some of these systems, such as LCLint, are meant to be static bug-finding tools. Like Cyclone, they typically require restricted coding idioms or additional annotations, but unlike Cyclone, they offer no soundness guarantees. This allows these static tools to minimize false positives. In contrast, Cyclone uses a combination of static analysis (for memory management) and dynamic checks (e.g., for buffer overruns) to minimize false positives.

Other systems, such as Safe-C and CCured do make safety guarantees by rewriting the code and adding additional run-time checks. The primary advantage of these systems is that they are (almost) completely transparent and require no changes to the 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. The disadvantages of this approach are (1) the storage for the object cannot be immediately recycled, and (2) a garbage collector is required. Furthermore, CCured may move some stack allocated objects to the heap to ensure that dangling-pointer dereferences can be caught.

Static Regions. Tofte and Talpin's seminal work [21] on implementing ML with regions provides the foundation for regions in the ML Kit [20]. Because ML programmers cede control over memory management, it is appropriate that the system performs region inference [19]. Nonetheless, effective programming can require some idioms to guide the inference [20]. More recent work for the ML Kit includes support for garbage collection within regions [13] as an alternative.

There have also been a number of extensions to the basic framework proposed by Tofte and Talpin, in order to avoid the constraints of LIFO region lifetimes. For instance, the ML Kit includes a reset-region primitive [20], Aiken et al. provided an analysis to free some regions early [1], Walker et al. [22, 23] have proposed general type 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 collectors *within* a language [24, 17]. 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. In the long run, we hope to incorporate simplified versions of these ideas in a future release of Cyclone. The primary challenge is, once again, to discover useful defaults that make programming and porting relatively convenient.

Two other recent projects, Vault [7] and the work of Henglein et al. [14] attempt 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 types can enforce that code *must* free a region. To do so, Vault must restrict region aliasing and track 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 for Cyclone.

Henglein et al. [14] have designed a flexible region system that does not require LIFO behavior. Essentially, a Floyd-Hoare Logic assigns a set of live regions to each program point. The primary advantage of this system is that it enjoys a "frame axiom" for reasoning about storage. 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. Like Cyclone, they provide convenient annotations to minimize the overheads of porting. One of these annotations, `parent_pointer`, can be used to simulate our notion of region subtyping. As a result, the RC approach is more flexible in the idioms that can be encoded.

However, RC does not guarantee that dangling pointers to stack-allocated objects are not dereferenced, nor does it guarantee that other kinds of type errors cannot occur. Indeed, we found a number of array bounds bugs in the benchmarks used to evaluate RC, such as `boyer`. Finally, RC cannot support the kind of polymorphism that Cyclone does because the compiler needs to statically know which objects are pointers.

In summary, the systems that are more flexible than Cyclone are either (a) unsound, (b) take control over memory management away from the programmer, (c) catch memory management errors at run-time, or (c) do not address important source-level issues such as minimizing annotations. This is not to say that Cyclone is better than these systems; ultimately, each has its strengths and weaknesses. Rather, we believe that Cyclone represents an interesting point in the design space. In the next section, we address what we believe are the most likely directions where we can profitably push the design.

8. FUTURE WORK

A great deal of work remains to achieve our ideal goal of providing a tool to easily move legacy code to a type-safe environment, and to provide 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, to handle the inter-procedural case, we expect to add support for explicit annotations, and to use experimental evidence to drive the choice of convenient defaults. Fortunately, we have designed our formal model with an eye towards these changes, so we expect our proof of soundness can be extended in a relatively straightforward fashion.

We also expect to incorporate better support for first-class regions, in the style of RC. Such regions will have weaker static guarantees and more run-time overhead, but will provide greater flexibility. 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.

Finally, though our type system has been formalized and proven sound, our type *inference* remains rather *ad hoc*. Right now, the inference engine is too eager to solve constraints, and sometimes rejects code that can be easily typed. For example, consider the following fragment:

```
region r1 {
  region r2 {
    int* x = rnew(r1) 0; // gives x type int*ρr1
    x = rnew(r2) 0;      // x needs type int*ρr2
 }
}
```

This code should type-check since `r1` outlives `r2`. In particular, by assigning `x` the type `int*`ρ_{r2}, the initializer is correct, since `int*`ρ_{r1} is a subtype of `int*`ρ_{r2}, as is the assignment on the next line. However, with the current inference engine, the code fails to check because the unifier eagerly pins `x`'s type to `int*`ρ_{r1}. Then the assignment on the second line fails because the value has type `int*`ρ_{r2} which cannot be promoted to `int*`ρ_{r1}.

The programmer can always avoid such problems by providing more type information (in our example, by filling in the missing region from `int*` to `int*`ρ_{r2}). Nonetheless, the unifier should be less eager by delaying the solution to constraints as in a traditional sub-typing framework.

In addition to enhancements to the region framework, a great deal of work is needed in other areas. For instance, right now, we see run-time overheads ranging anywhere from 1 to 10x that of the corresponding C code. We are currently working to identify the culprits, but an obvious problem is that we perform all array bounds checks at run-time. Fortunately, array-bounds-check elimination is a well-studied issue, and we hope to adapt a simple, but effective approach, such as the one used by CCured.

The other key area that we hope to push is data representations: To support dynamically sized arrays and array bounds checks, we must tag the array with implicit size information. Similarly, to support type-safe, tagged unions requires an implicit tag on the union. We are working to adapt the ideas from DML [26] and Xanadu [25] to provide programmer control over the placement of these tags. The hope is that 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 (&) as reported by Grossman [12].

9. ACKNOWLEDGMENTS

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

10. 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*, 2001. 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.

[13] N. Hallenberg. Combining garbage collection and region inference in the ML Kit. Master's thesis, Department of Computer Science, University of Copenhagen, 1999.

[14] 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*, page ???, Firenze, Italy, Sept. 2001.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

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

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.