Untrusted Garbage Collectors (Extended Abstract)

Daniel C. Wang Andrew W. Appel

Department of Computer Science
Princeton University
Princeton, NJ 08544 USA

July 17, 2000

Abstract

By combining existing type systems with several standard type-based compilation techniques, we describe how to write strongly typed programs that include a function that acts as a tracing garbage collector for the program. Since the garbage collector is an explicit function, we do not need to provide a *trusted* garbage collector as a runtime service to manage memory.

Since our language is strongly typed, the standard type soundness guarantee "Well typed programs do not go wrong" is extended to include the collector, making the garbage collector an *untrusted* piece of code. This is a desirable property for both Java and proof-carrying code systems. We describe the technique in detail and report performance measurements for a few microbenchmarks as well as sketch the proofs of type soundness for our system.

1 Introduction

Garbage collection is a forty-year-old technique for managing memory safely and efficiently. Garbage collection simplifies programming by removing the burden of deallocating data from the programmer. It makes programming interfaces simpler by hiding details of memory management. Since manual deallocation of heap objects is unsafe, garbage collection is the only practical way to manage heap allocation in safe programming languages such as ML, Scheme, and Java. The safety of all these languages is dependent on the correct implementation of a garbage collector. This makes the garbage collector a *trusted* component of the system.

In this paper we outline an approach, based on existing type systems, to build an *untrusted* garbage collector. An untrusted garbage collector is a collector whose safety is not assumed but statically verified. We can verify that the collector is not violating any invariants needed to guarantee type safety. The collector can now be an untrusted piece of code and is no longer part of the trusted computing base (TCB).

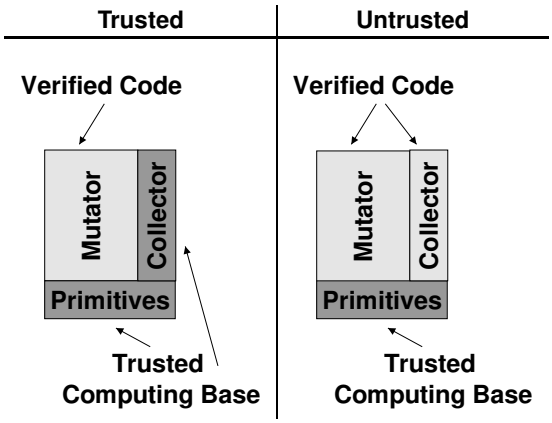

Figure 1: Trusted vs. Untrusted

This approach extends existing safety architectures, of Java byte-code verification and proof-carrying code, which statically verify safety properties of code provided by an untrusted *code producer* [19, 12]. We have extended this approach so we can verify the safety of the *mutator*[1] and collector. (See Figure 1)

Even if we are willing to trust the collector, formalizing the invariants needed to properly interface with the collector will add significant complexity to the safety policy in a proof carrying code system. Even *conservative garbage collectors*, which have significantly simpler interfaces, require the compiler to preserve certain invariants in order to guarantee safety [5].

[1] In the garbage collection literature, the mutator is the program whose storage is managed by the collector.

Our approach is simple: make the collector a well typed function written in the same language as the mutator. Garbage collection is no longer a primitive runtime service. Since the collector and mutator are both well typed, we simply prove type soundness of the language, and argue about the safety of the entire system by appealing to the fact that "Well typed programs do not go wrong." The important contribution of our work is the demonstration that even low-level runtime services such as garbage collectors can be assigned static types.

Being able to type-check our garbage collector has many advantages. It makes the system more secure, easier to debug, more flexible, and potentially more efficient. The garbage collector as well as its interface to the compiler is now statically verifiable. We can now catch some garbage collector bugs at compile time. If type information needs to be passed at runtime the garbage collector and compiler can use the same encoding of type information rather than having separate encodings and duplicate mechanisms for type dispatch. Code producers can provide a specialized collector to take advantage of special properties of their program, rather than relying on a fixed general purpose collector which may not be optimal for a given class of programs.

A Traditional Collector. Figure 2 provides a high-level description of a simple two-space stop-and-copy collector. When the collector is invoked it is passed three variables `from`, `k`, and `roots`, which are the current allocation space, the current continuation, and the set of live roots respectfully. Heap values are allocated in the current allocation space. The current continuation represents the "rest of the program" and takes as arguments an allocation space to allocate heap values and the *live roots* which represent all the currently reachable heap data the program may wish to use. All the data reachable from the live roots is allocated in the current allocation space.

The collector uses some heuristic to determine whether a garbage collection should occur. If a collection should occur the collector first creates a new fresh allocation space (`to`) then makes a deep copy of the live roots into the newly allocated *to-space*. All the data reachable from the new roots (`roots'`) should live in the new allocation space. The collector can now can safely free the old *from-space* and resume the program with the new allocation space and new live roots. Traditionally this operation is called a "flip" because once the from-space is deallocated its storage can immediately be reused as the next to-space, so the roles of the from-space and to-space are reversed.

Technical Challenges. Building a safe collector does not rely on a single key technical advance, but is results from the combination of several advances in typed compilation to build what is a standard garbage collector, with a surprisingly simple type system. The key issues that need to be addressed for this are:

1. Tracing objects of arbitrary type.
2. Primitive memory management.
3. Dealing with abstract objects such as closures.
4. Preserving pointer sharing in a safe and efficient way.

Because collectors must trace arbitrary values to identify reachable objects, they are difficult to type in traditional type systems found in languages like Java or ML. However, the work on *intensional type analysis* [13] provides a clean framework to address this difficulty as well as issues of how to pass type information at runtime [8]. If the type of every object is known at compile time we can apply standard *tagless* garbage collection techniques and generate specialized functions for each object of different type.

Another difficulty with collectors is that they use some more primitive memory management service to allocate and deallocate the from-space and the to-space. We must be able to verify that the more primitive service used by the collector is safe. The work on type and effect systems done by Tofte and Talpin and refined by others, provides type-safe explicit memory management [26, 1, 7, 4].

Traditional collectors violate data-abstraction guarantees that are present in the source language. So called "private fields" cannot remain private to the garbage collector. We can attempt to preserve these source level guarantees or make explicit what data-abstraction guarantees exist in a system when we take into account the garbage collector. We can rely on techniques used for type-preserving closure conversion to solve these problems [15, 27, 18].

Preserving pointer sharing is typically done by the use of *forwarding pointers* which provide an efficient way to implement a map from pointers in the from-space to pointers in the to-space. This map is needed to copy an arbitrary graph of heap objects from one space to the other. Any map, such as a hash-table, can be used in place of forwarding pointers. Dealing with forwarding pointers complicates reasoning about safety, but we outline one approach for dealing with forwarding pointers in a safe way.

Comparison to Region Inference. Our collector dynamically traces values at runtime, allowing for more fine-grain and efficient memory management compared to systems that use region inference, which may take asymptotically more space than a simple tracing garbage collector. From a different perspective, our untrusted garbage collector is merely a particular way of writing programs

```
fun gc(from, k, roots) =
 if(need_gc(from)) then
  let to = new_space() in
  let roots' = copy(from, to, roots)
  in free_space(from) ; k(to, root')
 else k(from, roots)
```

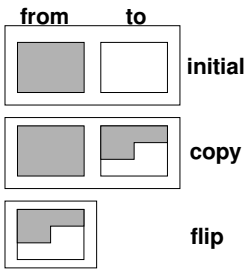

Figure 2: Traditional Garbage Collector

region context	Δ	::=	$\cdot \mid \Delta, \rho$
value context	Γ	::=	$\cdot \mid \Gamma, x : \sigma$
zero-order types	σ	::=	$C \mid \times(\vec{\sigma}) \mid +(\vec{\sigma}) \mid$
			$\mathsf{Ans} \mid (\sigma \ \mathsf{at}\ \rho) \mid \mathsf{fix}\, t.\sigma$
first-order types	ν	::=	$\forall[\Delta](\vec{\sigma}) \to \sigma$
terms	e	::=	$c \mid$
			$(\vec{e}) \mid \pi_i e \mid$
			$\mathsf{inj}_i^{\sigma} e \mid \mathsf{case}\ e\ \mathsf{of}\ \vec{a}\ \mathsf{end} \mid$
			$\mathsf{roll}^{\sigma} e \mid \mathsf{unroll}^{\sigma} e \mid$
			$\mathsf{halt}^{\sigma} \mid$
			$\mathsf{put}[\rho](e) \mid \mathsf{get}[\rho](e) \mid$
			$\mathsf{letr}\ \rho\ \mathsf{in}\ e \mid \mathsf{only}\ \Delta\ \mathsf{in}\ e \mid$
			$\mathsf{let}\ \mathsf{fix}\ \vec{d}\ \mathsf{in}\ e \mid \mathrm{f}[\Delta](\vec{e}) \mid$
			$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$
arm	a	::=	$(x : \sigma \Rightarrow e)$
decl	d	::=	$\mathrm{f}[\Delta](\Gamma) : \sigma = e$

Figure 3: Abstract Syntax of Core Calculus

in a language which uses regions as the primary memory management mechanism; with this perspective our work is simply a more efficient way of utilizing existing safe region-based memory management primitives, similar to the "double copying" technique used to make certain region programs more efficient [25]. This insight suggests that we can combine safe static memory management techniques with dynamic techniques to gain the benefit of both approaches.

In Section 2 we describe a simplified core calculus. In Section 3 we demonstrate our technique applied on a simple program. In Section 4 we discuss how to provide forwarding pointers in a type-safe way. Finally we present some preliminary performance numbers for a few microbenchmarks in Section 5.

2 A Language for Garbage Collectors

To simplify our presentation, we will only consider a first-order language where all types are known at compile time. This language is a restricted subset of ML with regions. There exist whole-program compilers for ML and Scheme that translate higher-order languages into a first-order language, so this restriction does not restrict the generality of our approach [16, 22].

Precise garbage collectors require that the compiler provide type information for the entire program. Garbage collectors have access to this whole program information when they are run, so our whole program assumption merely models the current state of the art. A technically more complicated framework will be needed to deal with situations where we wish to preserve separate compilation and retain all source-level abstraction guarantees. We consider these issues to be future work.

With these simplifying assumptions we can apply simple tagless techniques and generate a copy function for each type of object, rather than using the technically more sophisticated techniques of intensional type analysis. Since we are dealing with a first-order language, we do not need to deal with closures. The first-order assumption simplifies the presentation of the region system by allowing us to ignore *latent effects* [26]. It also forces compilers of higher-order languages targeting our first-order language to make the layout of closures explicit by representing closures as ordinary data structures [27]. These assumptions allow us to focus more of our attention on the underlying approach and some of the more problematic issues such as forwarding pointers.

Figure 3 describes the abstract syntax of our first-order language parameterized over base types C and constants c. All the constructs with the exception of the types and terms that deal with regions are standard. The zero-order types include n-ary products and sums as well as recursive types and a distinguished return type for continuations. The type $(\sigma\ \mathsf{at}\ \rho)$ describes values of type σ allocated in region ρ. The term $\mathsf{put}[\rho](e)$ allocates a value in

region ρ and the term get[ρ](*e*) reads values allocated in region ρ. The first-order type $\forall[\Delta](\vec{\sigma}) \rightarrow \sigma$ describes n-ary region-polymorphic functions that may refer only to region variables bound in the context Δ. We allow for the definition of mutually recursive functions and polymorphic recursion over region variables.

The term letr ρ in *e* introduces a new region variable whose scope does not escape the body of the expression. The term only Δ in *e* provides a method to deallocate a region early so we are not restricted to a simple LIFO region allocation policy present in the original Tofte-Talpin calculus. In order to safely deallocate a region early we require that the body of the only term not return, i.e., has type Ans. This is an important requirement that motivates our need to perform a CPS transformation on our source program. We must be able to typecheck the body of the expression using a subset (Δ) of the currently bound region variables. Regions not mentioned in Δ may be safe to deallocate, but we must perform some runtime checking to establish this fact. The runtime checking is needed to deal with issues of *region aliasing*. Unlike static approaches in previous systems we deal with aliasing issues by examining information about region aliasing at runtime [7].

Appendix A contains a specification of the static and dynamic semantics in the style of Wright and Felleisen [30] for the underlying region calculus. The region calculus itself is standard except for the method by which we deallocate regions early and deal with region aliasing. We will discuss this issue in more detail in Section 3.

To make these issues concrete we describe how to apply our technique to a simple program, iterative list reverse, written in our calculus using a more readable explicitly typed ML-like syntax.

3 Example: `itrev`

Source Program. Figure 4 contains a program that reverses a list of integers. The function `itrev` takes two arguments `l` and `acc` both of type `lst` and returns a value of type `lst`. The argument `l` holds the list to be reversed while `acc` holds the intermediate results. The recursive call to `itrev` is a tail call, so we do not need to maintain a stack in order to execute this code. As the function recursively descends `l` the previous list cells are garbage and can be reclaimed. The function therefore, uses a constant amount of live data in order to reverse a list of arbitrary length.

This simple reasoning about the space consumption cannot be applied in systems that use region inference to manage memory. Region inference would require that we keep a linear amount of live data. Type systems based on linear logic may give us more fine-grain control over allocation and deallocation and allow us to capture our reasoning for this particular instance, but they are fragile in the presence of aliasing [2, 3, 23, 29].

Consider the following variant of the program in Figure 4:

```
type lst = Nil | Cons (int, lst)
fun itrev(l:lst, acc:lst):lst = ...

let l = Cons(1, Cons(2, ... )) in
let rl = itrev(l, Nil)  in
let rl' = itrev(rl, Nil)
in (rl, rl')
```

Such a program would not be typeable in a linear framework because in the first call to `itrev` its first argument `l` is a unique occurrence while in the second call its first argument `rl` is no longer unique. In fact the list `rl` is live across the call to `itrev` so that the second call of `itrev` must hold on to all the cells of the list `rl` and cannot deallocate them early. We would like a memory management technique that can safely handle both calls to `itrev` and also reclaim the garbage cells in the first call of `itrev`. A technique that will handle both cases is tracing garbage collection, which will at runtime identify the set of potentially live objects.

We will show how to convert the source program in Figure 4 into a program that includes a function to garbage-collect dead values and more importantly preserve the fact that the program is well typed. In order to do this we will need to perform CPS and closure conversion to the program, to make our informal reasoning about the stack and live values explicit. Afterwards, we perform a simple region annotation to the resulting program to make precise what values live on the heap and when they are allocated. Finally, with this CPS-converted, closure-converted, region-explicit program we can synthesize a function that acts as a garbage collector for the program.

CPS and Closure Conversion. It simplifies our reasoning if we CPS convert our source program to make the control flow of the program explicit. However, since our language is first-order we cannot a standard CPS conversion algorithm, which would require the use of higher-order functions. Instead we adapt a first-order closure conversion technique outlined by Tolmach with a standard CPS conversion. Figure 5 illustrates Tolmach's technique. Notice that the types of any free variables are captured in the type of the closure [27].

Figure 6 is the result of applying these transformations on our example. Notice the new type `cont` which is the type of return continuations for the function `itrev`. In general each call site of `itrev` will require one new data constructor to represent each distinct return continuation.

```
type lst = Nil | Cons (int, lst)

fun itrev(l:lst, acc:lst):lst =
 case l of Nil ⇒ acc
 | Cons(hd,tl) ⇒
   let acc' = Cons(hd, acc)
   in itrev(tl, Cons(hd, acc'))

let l = Cons(1, Cons(2, ... )) in
let rl = itrev(l, Nil)
in rl
```

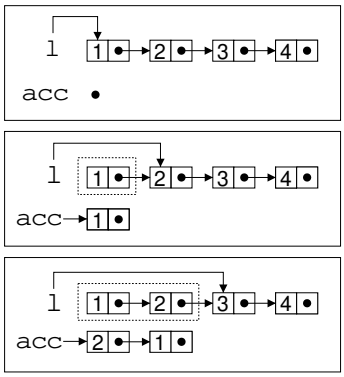

Figure 4: Iterative List Reverse

Higher-Order	First-Order

let y = 1 in
let f =
  if *e* then (λx:int.x)
  else (λx:int. x + y)
in f 1
	type clos = C1 | C2(int)
	fun apply (f, x) =
	  case f of C1 ⇒ x
	  | C2(y) ⇒ x + y
	let y = 1 in
	let f =
	  if *e* then C1
	  else C2(y)
	in apply (f, 1)

Figure 5: First-order Closure Conversion

```
type lst = Nil | Cons(int, lst)
type cont = Ret_rl

fun itrev(k:cont, l:lst, acc:lst):Ans = (* B *)
 case l of Nil ⇒ apply(k, acc) (* B1 *)
  | Cons(hd, tl) ⇒  (* B2 *)
    let acc' = Cons(hd, acc)
    in itrev(k, tl, acc')
and apply(k:cont, v:lst):Ans = (* C *)
  case k of Ret_rl ⇒ (* C1 *)
  let rl = v
  in rl ; halt()

let l = Cons(1, ...) in (* A *)
let k = Ret_rl
in itrev(k, l, Nil)
```

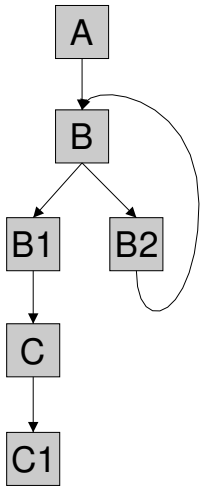

Figure 6: CPS-Converted and Closure-Converted Program

Also note that we implicitly assume we have access to the whole program at this point.

Many tagless garbage collection algorithms examine the return address of a function stored in the stack frame in order to determine the layout of the stack frames [9]. The transformation we have performed allows us to perform a similar operation. The tag of each data-constructor acts as the return address, the type of the data-constructor describes the stack layout, which is empty in this case.

Region Annotated. We have been informally arguing about where and when objects are allocated, Figure 7 shows our program with explicit region annotations. Notice that the type `lst` is now parameterized by a region in which the list lives. Since we can represent both the empty list and return continuation as single machine words we do not need to allocate space for them. We need to allocate space only when constructing list cells with the `Cons` data-constructor; this is reflected in the type of `Cons`, which will be allocated in the region ρ.

Both the `itrev` and `apply` function now take region parameters. When we allocate a new list cell we use the notation `lst[ρ].Cons(...)` to allocate a new list cell in the region ρ. We have assigned regions to types so that all values are allocated in one global region, which acts like a traditional heap. We could apply a more refined region analysis and avoid heap allocating an object when the lifetime of the object is statically obvious.

If the return continuation captured some live variables we would heap-allocate the continuation. This approach simplifies the compilation of advanced control features such as exceptions and call-cc as well as simplifying the reasoning of safety. However, heap-allocating return continuations could impact performance in an undesirable way. We believe we can adapt other techniques in the literature to avoid this penalty, but consider it to be future work.

GC Safe Points. Part of the interface between a garbage collector and the compiler is a description of "safe points". These are locations during the execution of the mutator where it is safe to invoke the garbage collector. At these safe points the compiler usually emits type information describing which values are live at the safe point. Compilers that do optimizations must also be careful not to perform certain optimizations across safe points. It is complicated to characterize precisely which optimizations are and are not allowed [9]. It requires modifying the compiler to understand the special semantics of what happens at a garbage-collection safe point.

In our framework all these issues are handled straightforwardly: since the garbage collector is just a normal function, the compiler does not need to be modified to be aware of any special semantics. A garbage collector is just a function that takes some data value. Figure 8 shows such a "safe point" in our program. Depending on some heuristic the code either continues executing or packages up the set of current live roots into a return continuation for the garbage collector, described by the type `gc_cont`.

With region types we are able to statically verify that the data value is actually the set of live roots for the entire program. If a buggy compiler or optimizer did not include all possible roots we would catch this error at compile time. More importantly, we would be able to identify what caused it which makes debugging significantly easier. Debugging these sorts of problems in a traditional unsafe system is considerably more difficult, because being able to isolate a bug of this sort in a large program is a serious challenge.

Early Deallocation and Region Aliasing. Figure 9 contains the code for the garbage collector. It copies the roots into a new region (ρ_{to}) then it *implicitly deallocates* the old region (ρ_{from}) and resumes the program with the new roots and new region. The term only ρ_{to} in ... is a static assertion that the body of the expression does not return, i.e. has type Ans, and can be safely evaluated using only the region bound to ρ_{to}. At runtime we simply note what region is dynamically bound to the region variable ρ_{to} and safely deallocate any other regions, since they are not needed to evaluate the rest of the program. The cost of this operation is linearly related to the number of live regions mentioned in the region context of the only expression. Our safe garbage collector needs at most two live regions at any time, so in practice the cost of this dynamic approach is negligible.

At first this may seem needlessly complex, but if we wish to have a general framework to deal with the early region deallocation we must perform this extra work at runtime to establish the safety of the deallocation. To see why this is the case consider the program

```
fun f[ρa, ρb](x:int at ρb):Ans =
 free_region ρa in
  get[ρb](x) ; halt()

letr ρ1, ρ2 in
 if e then f[ρ1, ρ2](put[ρ1](1))
 else f[ρ1, ρ1](put[ρ1](1))
```

Notice that if the program executes the first branch of the conditional then `f` behaves as expected. However, if we execute second branch then at runtime the region variables ρ_a and ρ_b are both bound to the same region variable ρ_1 and the program will attempt to access a region which we have erroneously deallocated. To handle this situation correctly we can simply prevent programs from aliasing

```
type lst[ρ] = Nil | Cons(int, lst[ρ]) at ρ
type cont[ρ] = Ret_rl

fun itrev[ρ](k:cont[ρ], l:lst[ρ], acc:lst[ρ]):Ans =
 case l of Nil ⇒ apply(k, acc)
  | Cons(hd, tl) ⇒
    let acc′ = lst[ρ].Cons(hd, acc)
    in itrev(k, tl, acc′)
and apply(k:cont[ρ], v:lst[ρ]):Ans = ...

letr ρ_heap in
let l = lst[ρ_heap].Cons(1, ...) in
let k = cont[ρ_heap].Ret_rl
in itrev[ρ_heap](k, l, lst[ρ_heap].Nil)
```

Figure 7: Program `itrev` after Region Annotation

```
type lst[ρ] = Nil | Cons(int, lst[ρ]) at ρ
type cont[ρ] = Ret_rl
type gc_cont[ρ] = Ret_itrev(cont[ρ], lst[ρ], lst[ρ]) at ρ

fun itrev[ρ](k:cont[ρ], l:lst[ρ], acc:lst[ρ]):Ans =
 if need_gc[ρ]() then  (*** safe point ***)
  let roots = gc_cont[ρ].Ret_itrev(k, l, acc)
  in gc[ρ](roots)
 else ...
and apply(k:cont, v:lst):Ans = ...
and gc[ρ_from](roots:gc_cont[ρ_from]):Ans = ...
...
```

Figure 8: Program `itrev` with Safe Point Inserted

```
type lst[ρ] = Nil | Cons(int, lst[ρ]) at ρ
type cont[ρ] = Ret_rl
type gc_cont[ρ] = Ret_itrev(cont[ρ], lst[ρ], lst[ρ]) at ρ

fun itrev[ρ](...):Ans = ... and apply(...) = ...
and gc[ρ_from](roots:gc_cont[ρ_from]):Ans =
 letr ρ_to in
  let roots′ = copy_gc_cont[ρ_from][ρ_to](roots) in
   only ρ_to in
    case roots′ of
     Ret_itrev(k, l, acc) ⇒ itrev[ρ_to](k, l, acc)
and copy_gc_cont[ρ1, ρ2](x:gc_cont[ρ1]):gc_cont[ρ2] = ...
...
```

Figure 9: "Flipping" from and to space

region variables through various typing disciplines [7]. The static approaches do not incur any runtime overhead, but are relatively complex systems and would disallow us from writing the program above.

However, if we are willing to perform a small amount of work at runtime we can rewrite `f` as

```
fun f[ρa, ρb](x:int at ρb):Ans =
 only ρb in
  get[ρb](x) ; halt()
```

At runtime we can determine what regions are actually bound to ρ_a and ρ_b. If ρ_a and ρ_b are bound to the same region we will deallocate nothing. If ρ_a and ρ_b are bound to distinct regions then we know that it is safe to deallocate the region associated with ρ_a since we do not need it to evaluate the rest of the computation. It is not hard to implement such a system in practice. In our prototype system all of the region primitives are less than 200 lines of C. We also believe that we can integrate the *explicit* deallocation techniques that use static typing to prevent region aliasing with our *implicit* approach to give us the benefits of both approaches, so that we resort to this dynamic approach when we are unable statically determine aliasing relationships.

This dynamic approach to region deallocation is similar to the work of Aiken and Gay [11]. However, they use a relatively weak region type system and a more expensive reference counting approach that requires updating a reference count for each interregion store. Because our type system provides more guarantees we can safely deallocate regions without needing to maintain any reference counts.

GC Copy Function. Figure 10 sketches the code for a naive copy function. The type of the copy function guarantees that the function performs a deep copy. The copy function is not written in continuation-passing style so it uses a stack while traversing the list. We could write the copy function in continuation-passing style and heap-allocate all its temporary space in a third region which we could reclaim after we are done. Alternatively if we extend our type system with enough technical machinery so that we can recycle the space used by the continuations we could implement what would amount to the Deutcsh-Schore-Waite pointer reversal algorithm [21, 28, 24]. A more serious problem with our copy function is that it does not preserve pointer sharing.

Preserving Sharing. Consider the data-structure in the first half of Figure 11. If we were to apply our garbage collection technique with a naive copy function it would convert the originally shared list of lists into an unshared version which uses more space. In the presence of cyclic

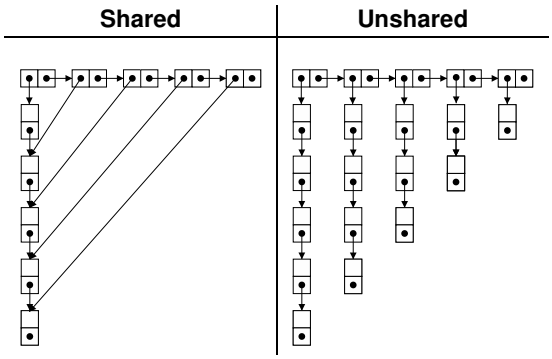

Figure 11: Shared vs. Unshared values

data structures our naive copy function would not terminate. Traditional garbage collectors use *forwarding pointers* to preserve sharing. However, forwarding pointers are not the only mechanism by which to do this.

Figure 12 outlines a copy function that uses an auxiliary hash table augmented with one primitive to return the unique pointer address of an object. This approach, while inefficient, demonstrates that the underlying algorithm needed to preserve sharing is not inherently difficult to type. In the next section we will outline how to encode forwarding pointers in a safe way.

4 Forwarding Pointers

The easiest way to understand how to encode forwarding pointers is to start by encoding as many of the garbage collector invariants as possible within the type system. We will discover that the type system outlined so far can capture many important invariants, but is not sufficiently expressive to capture them all precisely. However, if we examine our partial solution we will gain enough insight to come up with a full solution by extending our system with a single primitive.

Figure 13 sketches one approach to forwarding pointers. Some garbage collectors may overwrite a field of the object, but to simplify our presentation we assume every heap allocated object contains an extra word to hold a forwarding pointer which is either `NULL` or a pointer to an object of the appropriate type in the to-space. Notice that we have two different list types. The `gc_lst` type describes the garbage collector's view of lists. From the garbage collector's standpoint, lists are allocated in a from-space containing forwarding pointers into objects in a to-space. It must be the case that that lists allocated in the to-space have forwarding pointers which are always set to `NULL`. The `lst` type describes lists that the mutator operates on, and maintains the invariant that the forwarding pointer is set to `NULL`. The fact that the forwarding pointer is a mu-

```
type lst[ρ] = Nil | Cons(int, lst[ρ]) at ρ
type cont[ρ] = Ret_rl
type gc_cont[ρ] = Ret_itrev(cont[ρ], lst[ρ], lst[ρ]) at ρ

fun itrev[ρ](...):Ans = ... and apply(...) = ...
and gc[ρ_from](...):Ans = ...
and copy_gc_cont[ρ1, ρ2](x:gc_cont[ρ1]) =
 case x of Ret_itrev(k, l, acc) ⇒
  let k' = copy_cont[ρ1, ρ2](k) in
  let l' = copy_lst[ρ1, ρ2](l) in
  let acc' = copy_lst[ρ1, ρ2](acc)
  in gc_cont[ρ2].Ret_itrev(k', l', acc')
and copy_lst[ρ1, ρ2](x:lst[ρ1]):lst[ρ2] = ...
and copy_cont[ρ1, ρ2](x:cont[ρ1]):cont[ρ2] = ...
...
```

Figure 10: Copying roots

```
prim objId : [α] α → int
tycon tbl :: Rgn → Typ → Typ → Typ = ...
fun newTbl[ρ_tbl, α, β](sz:int):tbl[ρ_tbl, α, β] = ...
fun inTbl[ρ_tbl, α, β](t:tbl[ρ_tbl, α, β], key:α):bool = ...
fun getTbl[ρ_tbl, α, β](t:tbl[ρ_tbl, α, β], key:α):β = ...
fun addTbl[ρ_tbl, α, β](t:tbl[ρ_tbl, α, β], key:α, val:β):unit = ...

type lst[ρ] = Nil | Cons(int, lst[ρ]) at ρ
type cont[ρ] = Ret_rl
type gc_cont[ρ] = Ret_itrev(cont[ρ], lst[ρ], lst[ρ]) at ρ

fun itrev[ρ](...):Ans = ...
...
and share_copy_lst[ρ_tbl, ρ1, ρ2]
  (t:tbl[ρ_tbl, lst[ρ1], lst[ρ2]],x:lst[ρ1]):lst[ρ2] =
 case x of Nil ⇒ lst[ρ2].Nil
 | Cons(hd, tl) ⇒
    if inTbl[ρ_tbl, lst[ρ1], lst[ρ2]](x) then
     getTbl[ρ_tbl, lst[ρ1], lst[ρ2]](x)
     else let hd' = hd in
      let tl' = share_copy_lst[ρ_tbl, ρ1, ρ2](t,tl) in
      let x' = lst[ρ2].Cons(hd',tl')
      in addTbl[ρ_tbl, lst[ρ1], lst[ρ2]](x,x') ; x'
...
```

Figure 12: Preserving Sharing with a Hash-Table

```
tycon ref :: Rgn → Typ → Typ
type gc_lst[ρ_from, ρ_to] =  Nil
 | Cons(ref[ρ_from,fwd_ptr[ρ_to]], int, gc_lst[ρ_from, ρ_to]) at ρ_from
and fwd_ptr[ρ_to] = NULL | PTR(lst[ρ_to])
and lst[ρ_to] = Nil
 | Cons(ref[ρ_to, fwd_null], int, lst[ρ_to]) at ρ_to
and fwd_null = NULL
...
and share_copy_lst[ρ_from, ρ_to](x:gc_lst[ρ_from, ρ_to]):lst[ρ_to] =
 case x of Nil ⇒ lst[ρ_to].Nil
  | Cons(f, hd, tl) ⇒
    (case deref[ρ_from](f) of NULL ⇒
      let hd' = hd in
      let tl' = share_copy_lst[ρ_from, ρ_to] in
      let l = lst[ρ_to].Cons(mkref[ρ_to](fwd_null.NULL), hd' , tl')
      in f := l; l
     PTR(l) ⇒ l)
```

Figure 13: Encoding Forwading Pointers

table field which the garbage collector will mutate is captured by the use of the ref constructor.

The function `share_copy_lst` takes objects of type `gc_lst` and makes a copy of type `lst` which preserves the underlying pointer sharing in the original `gc_lst`. This code handles only acyclic lists but can be extended to handle the cyclic case in a natural way. At first glance this would seem to be a complete solution, unfortunately there is one thorny problem. If the mutator operates on objects of type `lst` how did we get an object of type `gc_lst` in the first place?

Ideally, we would like to argue that there is a natural subtyping relationship that allows us to coerce objects of type `lst` into objects of type `gc_lst`. For this to work we need the ref constructor to be covariant. However, it is well known that covariant references are unsound. However, Java adopts this rule for arrays[2] and achieves safety by requiring an extra runtime check for every array update. We cannot adopt the approach used by Java. This runtime check would prevent our garbage collector from setting any forwarding pointer to a non-null value.

However, rather than disallowing unsafe updates to an object we can do the dual and disallow unsafe dereferences, more importantly we can disallow unsafe dereferences in a way that does not require a runtime check for every access. Given a reference to a value of type `lst`, if after casting it to reference to a value of type `gc_lst` our program never accesses *any* value of type `lst` this cast is safe.

If our program is written in continuation-passing style, we can enforce this guarantee by making sure that after casting the value of type `lst` to a value of type `gc_lst` we pass the newly cast value immediately to a continuation that never accesses any value of type `lst`. One way to guarantee this condition statically is to type the continuation that receives the cast value in a typing context where the type `lst` is not bound.

Denying access to values of type `lst` after the program has performed a cast while safe, is too restrictive to be useful. However, since both the `lst` and `gc_lst` are region annotated types, we can achieve a similar sort of guarantee and still write useful programs by revoking the right to the access the region where the type `lst` is allocated, using a similar scoping trick. We can do this because after our garbage collector casts a `lst` value to a `gc_lst` value it never needs to examine the original value as a value of type `lst`. After our garbage collector runs, the original `lst` value is garbage, so the mutator never needs to access the region where the `lst` value was allocated. However, if we deny access to the type `lst` by denying access to the region it lives in, where is the value of type `gc_lst` allocated? We solve this problem by introducing a new "fake" region which is equivalent for the purposes of subtyping to the region we denied access to but for all practical purposes appears to be a distinct fresh region.

To do this we must introduce a non standard and ad-hoc form of subtyping on references. The allows for safe covariant references by using region variables to control access to potentially unsafe pointer aliases. Given two types `A` and `B` where `A` is a subtype of `B` and a region ρ the type ref`[`ρ`, A]` is a subtype of ref`[`ρ'`, B]` (where ρ' is a new "fake" region variable) provided that the rest of the program does not access any values in region ρ. This rule is admittedly ad-hoc, but it is the only ad-hoc rule in our entire system. Our approach is based on the observation

[2] A ref cell can be thought of as a one element array.

of Crary, Walker, et al [7] that region variables act like "capabilities". We use this observation to revoke all old references to the object and allow access to the object only through references of the object's supertype. Appendix A sketches the soundness of the approach for a simpler core calculus. It is important to note that we still must perform one more runtime alias check for this approach to be completely sound.

5 Practicality

The approach we have outlined so far is algorithmically competitive with existing garbage collection algorithms. There are obviously several important constant factors that we cannot neglect, if we wish to build a practical system. The first is code size. Since we are generating a new copy function for every unique type, code explosion is a serious concern. We can adapt the $\delta - main$ encoding technique [9] and other approaches to encourage sharing in our copy function to mitigate the code explosion problem. In order to address this issue we intend to do a detailed study of the number of unique copy functions needed for real programs. If sharing techniques are not sufficient we can adopt the techniques such as intensional type analysis [13] to avoid having to a distinct copy function for every unique type. For now we will ignore the issue of code size and just examine how close our approach is to a practical system, by running a few simple experiments.

Text for the submitted abstract

See Figures 14,15, and 16. We will include a more detailed discussion in the final paper that outlines our methodology in more detail, but omit it here because of space constraints. The summary is that the CPS conversion hurts us, because we are allocating more on the heap, so overall program performance is degraded.

However, if we estimate the per-word and per-object cost of a copy for both a unsafe DFS collector and our safe collector, we discover our tagless collector has significantly smaller per-object cost (24 cycles) compared to the unsafe collector (189 cycles), with a much higher per-word (16 cycles) cost compared to the unsafe collector (6 cycles). Since the per-object cost dominates the per-word cost, for small objects our collector is faster. The smaller per-object cost is due to our tagless approach that requires no interpretations of tags at runtime. The higher per-word cost can be explained because the system collector is able to use a optimized `memcpy` *function to copy objects, while our collector is copying objects word by word.*

We are using the `MLton` *compiler for our work [16]. Because of an experimental artifact, programs using our safe garbage collector do not benefit from certain unboxing optimizations that the compiler applies, which leads to higher program allocation in some cases, most noticeably in* **share-copy**. *Our safe collector using a hashtable allocates less space because it does not reserver space in each object for a forwarding pointer.*

Input Programs. For comparison we have chosen several programs seen in the previous literature on region based memory management. They are as follows:

ackermann Ackermann's function evaluated (n = 3, m = 6) [26]

appel1 Program designed to demonstrate issues of space complexity (n = 1000) [26]

appel2 Program designed to demonstrate issues of space complexity (n = 1000) [26]

fib Recursive Fibonacci (n = 33)

hsum Sum the value in a heap allocated list (n = 1000) [26]

inline Inline variant of appel1 (n = 1000) [26]

itrev Iterative list reverse (n = 10,000)

quicksort Quicksort randomly generated list (n = 1000) [26]

share-copy Reverse shared list of list (n = 10,000)

sum Recursive sum of the first n integers (n = 1000) [26]

start

(Text included only for final version) These programs are not a representative workload. However, they are sufficient for our preliminary evaluation. It is important to note that our safe collector for **appel1**, **appel2**, and **inline** uses asymptotically less space than a region based approach. Our safe collector is also more robust in that both **appel1** and **inline** have similar space characteristics which is not the case in the original Tofte-Talpin system.

Compiler. We have modified the back end of the `MLton` [16] to accept source programs that include a safe garbage collector. The `MLton` compiler emits C code that is then processes by the system C compiler to produce a runnable program. `MLton` has a straightforward unsafe depth-first-search two-space precise copying collector. The compiler also stack allocates activation records. *(End)*

end

We take each of our source programs and compile them in different ways to better understand the performance issues associated with our safe collector approach. The following list summarizes each variation:

orig Original program passed directly to `MLton`

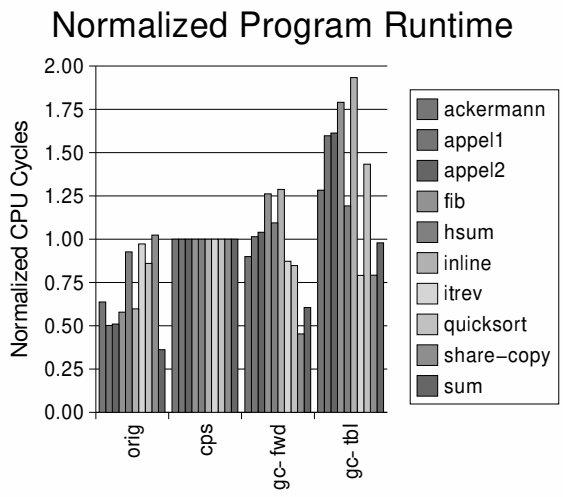

Figure 14: Relative Runtime Performance

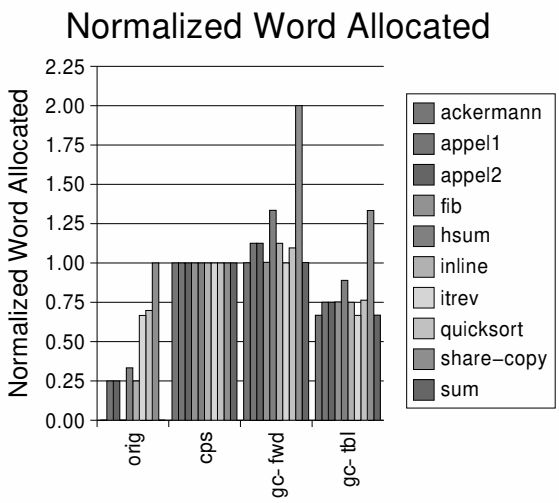

Figure 15: Relative Number of Words Allocated

cps Program run through CPS transform and first-order closure conversion, using `MLton`'s unsafe collector

gc-fwd Same as **cps** using safe collector and forwarding pointers which require an extra word of space for each object

gc-tbl Same as **cps** using safe collector with hashtable to preserver sharing

start

(Text included only for final version) We report numbers of the unaltered program passed directly to `MLton` and run with the default unsafe collector provided by `MLton`. To better understand the impact of CPS conversion, we measure the runtime of programs using the unsafe collector both before and after CPS conversion. We finally measure the performance of two different safe collectors, which differ only in their approach to sharing preservation, one uses a hashtable the other relies on forwarding pointers.

In a production system we would synthesize a safe collector after optimization, but because of the structure of `MLton` it was more convenient to synthesize a collector before optimization. However, this experimental artifact demonstrates that compiler backends can safely optimize our program after a garbage collector has been synthesized without understanding any special semantics. In this case there are two different optimizing compilers `MLton`, which is performing high-level optimizations such as inlining, record flattening, and unboxing, and the native C compiler.

Effect of CPS Conversion. Figure 14 shows the total wall-clock runtime for each program and variant normalized by the performance of the unoptimized *CPS converted* program. Immediately, one can see that the CPS conversion can cause more than a factor of two performance degradation when compared to the original program, which is stack allocating activation records. Figure 15 shows that our CPS converted programs are allocating significantly more heap data, which accounts for the performance difference. Also note that programs using a safe collector with a hashtable are allocating less data than programs using a safe collector with forwarding pointers. This is because although our safe collectors are tagless, we are reserving and extra word to store a forwarding pointer for each object. The unsafe collectors are paying a similar overhead for an extra tag word. The collector using a hashtable is not incurring this extra space overhead for tagging or forwarding. Unfortunately, synthesizing a garbage collector before optimizing the pervents `MLton` from performing certain space-saving optimizations, but this is simply an artifact of our current experimental setup. After we region annotate our program we make our allocation semantics explicit, and the `MLton` will not unbox objects which we have decided to box. This artifact most notably shows up in the increased allocation of **share-copy**.

If we compare the performance of CPS converted programs using our safe collector with the those using the standard unsafe collector, we see that in some instances programs using our safe collector seem to outperform the same programs using an unsafe collector. This naive comparison is misleading, because two different garbage collectors for the same program are not guaranteed to be invoked the same number of times or at the same safe points for identical programs. This explains why in the case of `itrev` our safe collector, which uses a more costly

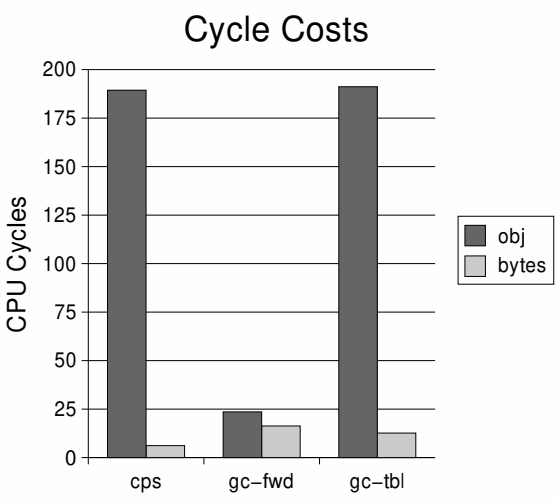

Figure 16: Per-word and per-object cycle costs

hashtable to preserv sharing, seems to outperform both the safe collector using forwarding pointers and the unsafe collector. Since each heap object is smaller when we are using a hashtable to preserve sharing, our collector will be invoked less frequently. In this case the program using the safe collector with a hashtable seems faster because it is just doing less work. We could perform an experiment where we control for this and force collections to occur at precisely the same time for identical programs, but this would obscure the fact that a garbage collection scheme which may be less efficient when comparing performance in terms of strict copying costs may in practice be more efficient because of secondary effects, such as reducing the object size overheads for the mutator.

Quantitative Experiment. Even with the caveat that raw copying performance is not an accurate measure of the performance impact of a garbage collection scheme, we have estimated the raw copying performance of our collector, by assuming the following:

$$gc\ time = c_1 \cdot objects\ collected + c_2 \cdot words\ collected$$

This assumes that total garbage collection time is simply the sum of time spent collecting each object and that the time spent collecting each object is simply some constant factor plus the cost collecting each word of the object. We have estimated the per-word and per-object costs by artificially varying the both the object size and number of objects collected for our set of programs and then performing a least squares fit over the data. Figure 16 summarizes our results in terms of absolute machine cycles. Since the unsafe collector is interpreting type tags at runtime it has a significantly higher per-object cost. However, it is using a system-optimized `memcpy` which allows it to have a much smaller per-word cost. Our tagless scheme allows us to avoid any tag-interpretation overhead. Our safe collector is copying objects with a series of naive loads and stores. For small objects, however, our safe collector using forwarding pointers is significantly more efficient than the unsafe collector. *(End)* **end**

Our experiments show that if we modify our framework so that we can stack-allocate return continuations and if code blowup is reasonable, we may end up with a faster collector, or at least a collector that is not significantly slower than a comparable unsafe collector.

6 Conclusion

Garbage collectors are typically written in low-level unsafe languages such as C. Most garbage collector algorithms discuss details in terms of low-level bit and pointer manipulation operations. Morrisett, Felleisen et al [17] present a high-level semantics for garbage collection algorithms, and prove the correctness of various well known algorithms. However, in their semantics garbage collection is still viewed as an abstract operation that lies outside of the underlying language being garbage collected. This approach allows them to discuss the purely algorithmic issues without revealing much if anything of the underlying implementation details. Our approach can be seen as a mid-level semantics for garbage collection where we reveal many of the pragmatic details need to actually implement the abstract algorithms in practice, but avoid bit-level details. Our semantics is sufficiently detailed enough that one can use it as guide to directly implement a reasonably efficient garbage collector on realistic hardware. It also has the property that we establish the safety or our garbage collection algorithm by simply relying on type soundness.

Although our approach as presented is not practical for general-purpose systems, we believe that we can extend our approach so that we can build practical systems with it. The most important insights are that a general-purpose garbage-collection system can be built on top of a set of much simpler primitives, and that when standard type systems are too weak, we can either rely on runtime checking or simply add "the right lemma" and encode what amounts to a small proof sublanguage to establish important pre-conditions needed for any ad-hoc reasoning that does not fit into a standard framework. An important contribution of our work is to demonstrate a particular combination of various approaches that we believe can be engineered into a practical and efficient system.

We would like to investigate how to integrate purely static memory management techniques [23, 29] with our system. Ideally we would like to have a spectrum of static

and dynamic memory management techniques so one can mix many memory management techniques in a clean, efficient, and safe way.

References

[1] Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of SIGPLAN'95 Conference on Programming Languages Design and Implementation*, volume 30 of *ACM SIGPLAN Notices*, pages 174–185, La Jolla, CA, June 1995. ACM Press.

[2] Henry G. Baker. Lively linear Lisp — 'Look Ma, no garbage!'. *ACM SIGPLAN Notices*, 27(9):89–98, August 1992.

[3] Henry G. Baker. The boyer benchmark meets linear logic. *Lisp Pointers*, 6(4):3–10, October 1993.

[4] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Region analysis and the polymorphic lambda calculus. In *Proceedings, Fourteenth Annual IEEE Symposium on Logic in Computer Science*, pages 88–97, Trento, Italy, 2–5 July 1999. IEEE Computer Society Press.

[5] Hans-Juergen Boehm. Simple garbage-collector safety. In *Proceedings of SIGPLAN'96 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 89–98. ACM Press, 1996.

[6] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33:309–338, 1998. Invited submission to special issue featuring a selection of contributions to the 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), 1997.

[7] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Conference Record of the Twenty-sixth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 262–275. ACM Press, 1999.

[8] Karl Crary, Stephaine Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In ICFP [14], pages 301–312.

[9] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Proceedings of SIGPLAN'92 Conference on Programming Languages Design and Implementation*, volume 27 of *ACM SIGPLAN Notices*, pages 273–282, San Francisco, CA, June 1992. ACM Press.

[10] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102:235–271, 1992. Original version in: Technical Report 89-100, Rice University, June 1989.

[11] David Gay and Alex Aiken. Memory management with explicit regions. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 313–323, Montreal, June 1998. ACM Press.

[12] J. Gosling. Java intermediate bytecodes. *ACM SIGPLAN Notices*, 30(3):111–118, March 1995.

[13] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of the Twenty-second Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages ??–?? ACM Press, January 1995.

[14] *Proceedings of Second International Conference on Functional Programming*, Baltimore, MA, September 1998.

[15] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Conference Record of the Twenty-third Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 271–283. ACM Press, 1996.

[16] MLton, a whole program optimizing compiler for Standard ML. `http://www.neci.nj.nec.com/PLS/MLton/`.

[17] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Functional Programming and Computer Architecture*, San Diego, 1995.

[18] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In POPL [20], pages 85–97.

[19] George Necula. Proof-carrying code. In *Conference Record of the Twenty-fourth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 106–119. ACM Press, 1997.

[20] *Conference Record of the Twenty-fifth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, 1998.

[21] H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[22] Jeffrey Mark Siskind. Flow-directed lightweight closure conversion. Technical Report TR99–190R, NEC Reseaarch Institute, Inc., December 1999. inpreparation.

[23] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In Gert Smolka, editor, *Ninth European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, Berlin, April 2000. Springer-Verlag.

[24] Jonathan Sobel and Daniel P. Friedman. Recycling continuations. In ICFP [14], pages 251–270.

[25] Mads Tofte, Lars Birkedal, Martin Elsman, Neils Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with reigons in the ML Kit (for version 3). Technical Report DIKU-TR-98/25, University of Copenhagen, December 1998.

[26] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 188–201. ACM Press, January 1994.

[27] Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

[28] G. Veillon. Transformations de programmes recursifs. *R.A.I.R.O. Informatique*, 10(9):7–20, September 1976.

[29] David Walker and Greg Morrisett. Alias types for recursive data structures (extended version). Technical Report TR2000-1787, Cornell University, March 2000.

[30] Wright and Felleisen. A syntactic approach to type soundness. *Information and Computation (formerly Information and Control)*, 115, 1994.

A Formal Semantics

To clarify each language feature we present several different calculi that we progressively modify and refine until we finally have a calculus that includes regions, a form of structural subtyping, references, and finally safe covariant references. There is a type soundness theorem for each calculus. The semantics are relatively standard. Our encoding of subtyping is based on the work of Brandt and Henglein [6], which provides us a framework with more control over what contexts certain subtyping rules are applied. This control is important if we wish to encode safe covariant references.

We have attempted to keep our notation standard. However, it simplifies the presentation if we use the $\uplus$ operator which represents this disjoint union of two sets, or the union of two finite functions when the domain of the two functions are two disjoint sets. We also consider alpha convertible terms equivalent and assume all bound variables are unique and renamed by substitution to avoid capture. The notation $\vec{s}$ abbreviates a sequence of zero or more syntactic forms $\vec{s_1} \cdot \vec{s_2}$ abbreviates the concatenation of two sequences.

A.1 Core Region Calculus

The first-order restriction to our language is important if we are using it as a practical compilation calculus, but it clutters the presentation of our semantics. We can simplify the presentation if we assume the first-order calculus is simply a syntactic subset of a higher-order calculus.

$$
\begin{array}{rcl}
\Delta & ::= & \{\} \mid \{\rho_1, \ldots \rho_n\} \\
\Gamma & ::= & \cdot \mid \Gamma, x : \sigma \\
\sigma & ::= & C \mid \sigma_1 \xrightarrow{\Delta} \sigma_2 \mid \forall \rho.\sigma \mid (\sigma \text{ at } \rho) \mid \mathsf{Ans} \\
e & ::= & x \mid c \mid (\lambda x : \sigma.e) \mid (e_1\ e_2) \mid \\
 & & (\Lambda \rho.e) \mid e[\rho] \mid \\
 & & \mathsf{letr}\ \rho\ \mathsf{in}\ e \mid \mathsf{only}\ \Delta\ \mathsf{in}\ e \mid \\
 & & \mathsf{put}[\rho](e) \mid \mathsf{get}[\rho](e) \mid \\
 & & \mu f : \sigma.e \mid \mathsf{halt}^{\sigma} \\
v & ::= & c \mid (\lambda x : s.e) \mid (\Lambda \rho.e) \mid \mathsf{put}[\rho](v) \mid \\
 & & \mathsf{halt}^{\sigma}
\end{array}
$$

Since our calculus is higher-order we must account for latent effects. In our first-order calculus the type

$$\forall[\Delta](\sigma_1, \ldots, \sigma_i) \to \sigma$$

abbreviates the higher-order type

$$\forall \rho_1 \ldots \forall \rho_n.\sigma_1 \xrightarrow{\Delta} \ldots \sigma_i \xrightarrow{\Delta} \sigma \text{ where } \Delta = \rho_1, \ldots, \rho_n$$

$$\frac{}{\Delta;\Gamma,x:\sigma\vdash x:\sigma}\text{t_var} \qquad \frac{}{\Delta;\Gamma\vdash c:C}\text{t_const}$$

$$\frac{\Delta';\Gamma,x:\sigma\vdash e:\sigma_2}{\Delta\uplus\Delta';\Gamma\vdash(\lambda x:\sigma.e):\sigma_1\xrightarrow{\Delta'}\sigma_2}\text{t_lam}$$

$$\frac{\Delta\uplus\Delta';\Gamma\vdash e_1:\sigma_1\xrightarrow{\Delta'}\sigma_2 \qquad \Delta\uplus\Delta';\Gamma\vdash e_2:\sigma_2}{\Delta\uplus\Delta';\Gamma\vdash(e_1\ e_2):\sigma_1}\text{t_app}$$

$$\frac{\Delta\uplus\{\rho\};\Gamma\vdash e:\sigma}{\Delta;\Gamma\vdash(\Lambda\rho.e):\forall\rho.\sigma}\text{t_rho_abs} \qquad \frac{\Delta\uplus\{\rho'\};\Gamma\vdash e:\forall\rho.\sigma}{\Delta\uplus\{\rho'\};\Gamma\vdash e[\rho']:\sigma[\rho'/\rho]}\text{t_rho_app}$$

$$\frac{\Delta\uplus\{\rho\};\Gamma\vdash e:\sigma \qquad \{\rho\}\cap FRV(\sigma)=\{\}}{\Delta;\Gamma\vdash\mathsf{letr}\ \rho\ \mathsf{in}\ e:\sigma}\text{t_letr}$$

$$\frac{\Delta\uplus\Delta';\Gamma\vdash e:\mathsf{Ans}}{\Delta\uplus\Delta';\Gamma\vdash\mathsf{only}\ \Delta'\ \mathsf{in}\ e:\mathsf{Ans}}\text{t_only}$$

$$\frac{\Delta\uplus\{\rho\};\Gamma\vdash e:\sigma}{\Delta\uplus\{\rho\};\Gamma\vdash\mathsf{put}[\rho](e):(\sigma\ \mathsf{at}\ \rho)}\text{t_put} \qquad \frac{\Delta\uplus\{\rho\};\Gamma\vdash e:(\sigma\ \mathsf{at}\ \rho)}{\Delta\uplus\{\rho\};\Gamma\vdash\mathsf{get}[\rho](e):\sigma}\text{t_get}$$

$$\frac{\Delta;\Gamma,f:\sigma\vdash e:s}{\Delta;\Gamma\vdash(\mu f:\sigma.e):\sigma}\text{t_fix} \qquad \frac{}{\Delta;\Gamma\vdash\mathsf{halt}^\sigma:\sigma}\text{t_halt}$$

$$\begin{array}{lcl} E & ::= & [\,] \mid (E\ e) \mid (v\ E) \mid E[\rho] \mid \mathsf{put}[\rho](E) \mid \mathsf{get}[\rho](E) \\ P & ::= & \langle\Delta,\ e\rangle \\ A & ::= & \langle\Delta,\ v\rangle \end{array}$$

$$\begin{array}{lrcl}
\text{r_app} & ((\lambda x:\sigma.e)\ v) & \overset{\text{pure}}{\longmapsto} & e[v/x] \\
\text{r_rho_app} & (\Lambda\rho.e)[\rho'] & \overset{\text{pure}}{\longmapsto} & e[\rho'/\rho] \\
\text{r_fix} & \mu\, f:\sigma.e & \overset{\text{pure}}{\longmapsto} & e[\mu\, f:\sigma.e/f] \\
\\
\text{r_pure} & \langle\Delta,\ E[e]\rangle & \longmapsto & \langle\Delta,\ E[e']\rangle \text{ where } e \overset{\text{pure}}{\longmapsto} e' \\
\text{r_halt} & \langle\Delta,\ E[\mathsf{halt}^\sigma]\rangle & \longmapsto & \langle\Delta,\ \mathsf{halt}^\sigma\rangle \\
\text{r_get_put} & \langle\Delta\uplus\{\rho\},\ E[\mathsf{get}[\rho](\mathsf{put}[\rho](v))]\rangle & \longmapsto & \langle\Delta\uplus\{\rho\},\ E[v]\rangle \\
\text{r_letr} & \langle\Delta,\ E[\mathsf{letr}\ \rho\ \mathsf{in}\ e]\rangle & \longmapsto & \langle\Delta\uplus\{\rho\},\ E[e]\rangle \\
\text{r_only} & \langle\Delta\uplus\Delta',\ E[\mathsf{only}\ \Delta'\ \mathsf{in}\ e]\rangle & \longmapsto & \langle\Delta',\ e\rangle \\
\text{r_dealloc} & \langle\Delta\uplus\{\rho\},\ e\rangle & \longmapsto & \langle\Delta,\ e\rangle \text{ where } \rho\notin FRV(e)
\end{array}$$

Figure 17: Semantics of Core Calculus

Terms are related in the natural way.

Figure 17 contains the static dynamic semantics for our core calculus in the style of Wright and Felleisen [30]. The judgment

$$\Delta;\Gamma \vdash e : \sigma$$

states that under region context Δ and typing environment Γ expression e has type σ. We implicitly assume that Γ is a well formed environment such that every free region variable that occurs in Γ is bound in Δ. Programs consist of a region context Δ which models the set of live allocated regions in the program. The precise definition of free region variables is given below:

$$\begin{aligned}
FRV(\sigma_1 \xrightarrow{\Delta} \sigma_2) &= FRV(\sigma_1) \cup FRV(\sigma_2) \cup \Delta \\
FRV(\forall \rho.\sigma) &= FRV(\sigma) \setminus \{\rho\} \\
FRV((\sigma \text{ at } \rho)) &= FRV(\sigma) \cup \{\rho\} \\
FRV(\mathsf{Ans}) &= \{\} \\
FRV(C) &= \{\} \\
\\
FRV(x) &= \{\} \\
FRV(c) &= \{\} \\
FRV((\lambda x : \sigma.e)) &= FRV(e) \cup FRV(\sigma) \\
FRV((e_1\ e_2)) &= FRV(e_1) \cup FRV(e_2) \\
FRV((\Lambda \rho.e)) &= FRV(e) \setminus \{\rho\} \\
FRV(e[\rho]) &= FRV(e) \cup \{\rho\} \\
FRV(\mathsf{letr}\ \rho\ \mathsf{in}\ e) &= FRV(e) \setminus \{\rho\} \\
FRV(\mathsf{only}\ \Delta\ \mathsf{in}\ e) &= FRV(e) \cup \Delta \\
FRV(\mathsf{put}[\rho](e)) &= FRV(e) \cup \{\rho\} \\
FRV(\mathsf{get}[\rho](e)) &= FRV(e) \cup \{\rho\} \\
FRV(\mu\, f : \sigma.e) &= FRV(e) \cup FRV(\sigma) \\
FRV(\mathsf{halt}^{\sigma}) &= FRV(\sigma)
\end{aligned}$$

The definition of free region variables is straightforward. Our rules make it clear that the region context in the used by term only does not bind any new region variables. For simplicity we assume base types and constants do not contain any region variables.

Our semantics includes a general region deallocation rule (r_dealloc) for unused regions which provides an abstract way to reclaim dead regions after the evaluation of the body of a letr ρ in e expression. This general rule may seem to make the rule r_only unnecessary, but notice that the body of the r_only rule is known not to return. This fact allows us to safely throw away the useless surrounding control context and associated useless region variables which the r_dealloc rule would not normally reclaim. Other than the special terms for regions the calculus is a standard call-by-value typed lambda calculus.

The relation $\longmapsto^*$ is the reflexive transitive closure of the one step reduction relation $\longmapsto$. A well typed program is in a *stuck* state if from P there does not exist P' such that $P \longmapsto P'$ and P is not an answer. The calculus presented so far and all of its future extensions enjoy the following type soundness property.

Theorem 1 (Type Soundness) *If* $\Delta;\Gamma \vdash P : \sigma$ *and* $P \longmapsto^* P'$ *then* P' *is not a stuck state.*

Proof: By Type Preservation and Progress Lemmas.

Lemma 1.1 (Type Preservation) *If* $\Delta;\Gamma \vdash e : \sigma$ *and* $\langle \Delta, e \rangle \longmapsto \langle \Delta', e' \rangle$ *then* $\Delta';\Gamma \vdash e' : \sigma$

Proof: Standard

Lemma 1.2 (Progress) *If* $\Delta;\Gamma \vdash P : \sigma$ *then either:*

- *There exists* P' *such that* $P \longmapsto P'$*, or*
- P *is a answer.*

Proof: Standard

A.2 Coercion Based Subtyping

Starting from the core calculus we introduce n-ary sums and subtyping on those terms through explicit coercions. The n-ary sums are encoded in a natural way. Our subtyping rules are based on the work of Brandt and Henglein [6]. We provided a separate language of coercion functions which act as constructive proofs of a subtyping relationship, between two types. Our modified language is as follows:

$$\begin{aligned}
\sigma &::= \ldots \mid +(\vec{\sigma}) \\
e &::= \ldots \mid \mathsf{inj}_i^{\sigma} e \mid \mathsf{case}\ e\ \mathsf{of}\ \vec{e}\ \mathsf{end} \mid \\
&\quad\ \mathsf{coerce}[\![cp]\!](e) \\
v &::= \ldots \mid \mathsf{inj}_i^{\sigma} v \\
\gamma &::= \cdot \mid \gamma, x_{cp} \triangleright cf \\
cf &::= \sigma_1 \leq \sigma_2 \\
cp &::= x_{cp} \mid \mathsf{refl}[\sigma] \mid \\
&\quad\ \mathsf{trans}[\sigma_1, \sigma_2, \sigma_3](cp_1, cp_2) \mid \\
&\quad\ \mathsf{cut}\, x_{cp} : cf \ = \ cp_1\ \mathsf{in}\ cp_2 \mid \\
&\quad\ \mathsf{sum}[\sigma_1, \sigma_2]
\end{aligned}$$

None of the new terms bind new region variables so the definition of free region variables is extended in the obvious way.

Figure 18 extends the previous static and dynamic semantics to account for these new features. The cp terms form a language of coercion proofs which are used to

$$\ldots$$

$$\frac{\Delta;\Gamma \vdash e : \sigma_i}{\Delta;\Gamma \vdash \mathsf{inj}_i^{+(\vec{\sigma})} e : +(\vec{\sigma})}\,\mathsf{t_inj_i}$$

$$\frac{\Delta;\Gamma \vdash e : +(\vec{\sigma}) \qquad \text{forall } e_i\ \Delta;\Gamma \vdash e_i : \sigma_i \xrightarrow{\Delta} \sigma' \qquad \vec{\sigma} = \sigma_1, \ldots, \sigma_n}{\Delta;\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \vec{e}\ \mathsf{end} : \sigma'}\ \mathsf{t_case}$$

$$\frac{\Delta;\cdot \vdash cp \triangleright \sigma \leq \sigma' \qquad \Delta;\Gamma \vdash e : \sigma}{\Delta;\Gamma \vdash \mathsf{coerce}[\![cp]\!](e) : \sigma'}\,\mathsf{t_coerce}$$

$$\frac{}{\Delta;\gamma, x \triangleright cf \vdash x_{cp} \triangleright cf}\,\mathsf{cp_var} \qquad \frac{}{\Delta;\gamma \vdash \mathsf{refl}[\sigma] \triangleright \sigma \leq \sigma}\,\mathsf{cp_refl}$$

$$\frac{\Delta;\gamma \vdash cp_1 \triangleright \sigma_1 \leq \sigma_2 \qquad \Delta;\gamma \vdash cp_2 \triangleright \sigma_2 \leq \sigma_3}{\Delta;\gamma \vdash \mathsf{trans}[\sigma_1,\ \sigma_2,\ \sigma_3](cp_1,\ cp_2) \triangleright \sigma_1 \leq \sigma_3}\,\mathsf{cp_trans}$$

$$\frac{\Delta;\gamma \vdash cp_1 \triangleright cf \qquad \Delta;\gamma, x_{cp} \triangleright cf \vdash cp_2 \triangleright cf'}{\Delta;\gamma \vdash \triangleright \mathsf{cut}\, x_{cp} : cf\ =\ cp_1\ \mathsf{in}\ cp_2 cf'}\,\mathsf{cp_cut}$$

$$\frac{\vec{\sigma_1} \cdot \vec{\sigma'} = \vec{\sigma_2}}{\Delta;\gamma \vdash \mathsf{sum}[+(\vec{\sigma_1}),\ +(\vec{\sigma_2})] \triangleright +(\vec{\sigma_1}) \leq +(\vec{\sigma_2})}\,\mathsf{cp_sum}$$

$$E \quad ::= \quad \ldots \mid \mathsf{inj}_i^{\sigma} E \mid \mathsf{case}\ E\ \mathsf{of}\ \vec{e}\ \mathsf{end} \mid \mathsf{coerce}[\![cp]\!](E)$$

$$\begin{array}{lrcl}
 & & \ldots & \\
\mathsf{r_case} & \mathsf{case}\ \mathsf{inj}_i^{\sigma} v\ \mathsf{of}\ \vec{e}\ \mathsf{end} & \xrightarrow{\text{pure}} & (e_i\ v) \\
\mathsf{r_refl} & \mathsf{coerce}[\![\mathsf{refl}[\sigma]]\!](v) & \xrightarrow{\text{pure}} & v \\
\mathsf{r_trans} & \mathsf{coerce}[\![\mathsf{trans}[\sigma_1,\ \sigma_2,\ \sigma_3](cp_1,\ cp_2)]\!](v) & \xrightarrow{\text{pure}} & \mathsf{coerce}[\![cp_2]\!](\mathsf{coerce}[\![cp_1]\!](v)) \\
\mathsf{r_cut} & \mathsf{coerce}[\![\mathsf{cut}\, x_{cp} : cf\ =\ cp_1\ \mathsf{in}\ cp_2]\!](v) & \xrightarrow{\text{pure}} & \mathsf{coerce}[\![cp_2[cp_1/x_{cp}]]\!](v) \\
\mathsf{r_sum} & \mathsf{coerce}[\![\mathsf{sum}[\sigma_1,\ \sigma_2]]\!](\mathsf{inj}_i^{\sigma_1} v) & \xrightarrow{\text{pure}} & \mathsf{inj}_i^{\sigma_2} v \\
 & & \ldots &
\end{array}$$

Figure 18: Calculus with Subtyping

prove coercion formulas, cf, of the form $\sigma_1 \leq \sigma_2$ that is σ_1 is a subtype of σ_2 or values of type σ_1 can be coerced to values of type σ_2.

We introduce a new judgment

$$\Delta;\gamma \vdash cp \triangleright cf$$

that determines whether a given cp is a valid proof for a given coercion formula under assumptions γ and context Δ. In our semantics we have provided a simple operational interpretation to our coercion functions, as suggested by Brant and Henglein but in a realistic system the coercion functions would have no operational significance.

We again implicitly assume that γ is well formed with respect to Δ. By controlling the initial assumptions in γ we can control under what contexts certain subtyping judgments are allowed. This control will be critical for our encoding of covariant references later.

To keep the presentation simple we have not included a complete set of subtyping proof terms. We do include a cut term to allow for more convenient proofs even though it could be eliminated from our calculus. The cp_cut rule also illustrates a non-trivial use of our subtyping context γ. We provide a single proof term for sums (sum) which allows for "width subtyping" over sums. In our semantics sum values are explicitly tagged with the type that they belong too. In order for our semantics to guarantee type preservation when we coerce a value with a sum proof term we must adjust the type of the value appropriately. This is purely notational and would have no effect in a real implementation.

A.3 References

References are introduced in the style of Felleisen and Hieb [10]. We modify their presentation to account for the fact that each reference is allocated in a particular region. The new extended syntax is as follows:

$$\begin{array}{rcl}
\theta & ::= & \{\} \mid \{\rho_1 \cdot x_1 : \sigma \mapsto v_1, \ldots, \\
& & \qquad \rho_n \cdot x_n : \sigma_n \mapsto v_n\} \\
\sigma & ::= & \ldots \mid \mathsf{ref}[\rho, \sigma] \\
e & ::= & \ldots \mid \\
& & \mathsf{st}\ \theta\ \mathsf{in}\ e \mid \rho \cdot x \mid \\
& & \mathsf{mkref}^{\sigma}[\rho](e) \mid \mathsf{deref}^{\sigma}[\rho](e) \mid \\
& & \mathsf{upd}^{\sigma}[\rho](e_1, e_2); e_3 \mid \\
v & ::= & \ldots \mid \rho \cdot x
\end{array}$$

None of the new terms bind new region variables so the definition of free region variables is extended in the obvious way.

A store is represented as a map from references to values. A reference $\rho \cdot x$ is a unique label which can be decomposed into a region tag and some other unique label. One may think of the region tag as being the higher-order bits of a pointer address with the remaining lower-order bits making up the remaining address.

Figure 19 contains the semantics for our new language. The new judgment

$$\Delta;\Gamma \vdash \theta\ \text{st-ok}$$

determines whether a store forms a consistent set of bindings. Since the bindings in stores are mutually recursive we explicitly type each binding to simplify checking. Programs now consists of a region context and top-level global store. Notice unlike Morrisett and Felleisen et al [17] we do not include a garbage collection rule to remove unneeded bindings from our store, but modify several of the deallocation rules for regions to also deallocate useless bindings from the global store.

A.4 Covariant References

To allow for subtyping of references we modify our definition of a store to be a mapping from region tagged labels to values that may have several coercions applied to them. One effect of this change is that when we dereference an address we may have to evaluate several coercions before we are left with a value, since coercions are of only notational importance the effect of the lazy coercions do not complicate a realistic implementation. We also extend our language with a new term gc_lemma, which allows for covariant references in a restricted context. The new extend syntax is as follows:

$$\begin{array}{rcl}
\theta & ::= & \{\} \mid \{\rho_1 \cdot x_1 : \sigma \mapsto cv_1, \ldots, \\
& & \qquad \rho_n \cdot x_n : \sigma_n \mapsto cv_n\} \\
e & ::= & \ldots \mid \mathsf{gc_lemma} \\
& & \quad \mathsf{kill}\ \rho_1\ \mathsf{resurrect\ as}\ \rho_2 \\
& & \quad \mathsf{assuming}\ cp_1\ \mathsf{proves}\ x_{cp} : cf \\
& & \quad \mathsf{let}\ x\ =\ \mathsf{coerce}[\![cp_2]\!](e_1)\ \mathsf{in} \\
& & \quad \mathsf{only}\ \Delta\ \mathsf{in}\ e_2 \\
cv & := & \mathsf{coerce}[\![cp]\!](cv) \mid v
\end{array}$$

The the second region variable of the new term gc_lemma is a binding occurrence the so we extend the notion of free region variables as follows:

$$\begin{array}{l}
FRV(\mathsf{gc_lemma} \\
\quad \mathsf{kill}\ \rho_1\ \mathsf{resurrect\ as}\ \rho_2 \\
\quad \mathsf{assuming}\ cp_1\ \mathsf{proves}\ x_{cp} : cf \\
\quad \mathsf{let}\ x\ =\ \mathsf{coerce}[\![cp_2]\!](e_1)\ \mathsf{in} \\
\quad \mathsf{only}\ \Delta\ \mathsf{in}\ e_2) = \\
\quad (\{\rho_1\} \cup FRV(cf) \cup FRV(cp_1) \cup FRV(cp_2) \cup \\
\qquad FRV(e_1) \cup FRV(\Delta) \cup FRV(e_2)) \setminus \{\rho_2\}
\end{array}$$

...

$$\frac{}{\Delta;\Gamma \vdash \{\}\ \text{st-ok}}\,\text{st_ok_empty} \qquad \frac{\Delta \uplus \{\rho\};\Gamma \vdash v : \sigma \qquad \Delta \uplus \{\rho\};\Gamma \vdash \theta\ \text{st-ok}}{\Delta \uplus \{\rho\};\Gamma \vdash \theta \uplus \rho \cdot x : \sigma \mapsto v\ \text{st-ok}}\,\text{st_ok_vbind}$$

$$\begin{array}{rcl} J_1 & = & \Delta;\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash e : \sigma \\ J_2 & = & \Delta;\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash \{\rho_1 \cdot x_1 : \sigma_1 \mapsto v_1, \ldots \rho_n \cdot x_n : \sigma_n \mapsto v_n\}\ \text{st-ok} \end{array}$$

$$\frac{J_1 \qquad J_2}{\Delta;\Gamma \vdash \mathsf{st}\ \{\rho_1 \cdot x_1 : \sigma_1 \mapsto v_1, \ldots \rho_n \cdot x_n : \sigma_n \mapsto v_n\}\ \mathsf{in}\ e : \sigma}\,\text{t_store}$$

$$\frac{}{\Delta \uplus \{\rho\};\Gamma, x : \sigma \vdash \rho \cdot x : \mathsf{ref}[\rho,\ \sigma]}\,\text{t_ref} \qquad \frac{\Delta \uplus \{\rho\};\Gamma \vdash e : \sigma}{\Delta \uplus \{\rho\};\Gamma \vdash \mathsf{mkref}^{\sigma}[\rho](e) : \mathsf{ref}[\rho,\ \sigma]}\,\text{t_mkref}$$

$$\frac{\Delta \uplus \{\rho\};\Gamma \vdash e : \mathsf{ref}[\sigma,\ \rho]}{\Delta \uplus \{\rho\};\Gamma \vdash \mathsf{deref}^{\sigma}[\rho](e) : \sigma}\,\text{t_deref}$$

$$\frac{\Delta \uplus \{\rho\};\Gamma \vdash e_1 : \mathsf{ref}[\sigma,\ \rho] \qquad \Delta \uplus \{\rho\};\Gamma \vdash e_2 : \sigma \qquad \Delta \uplus \{\rho\};\Gamma \vdash e_3 : \sigma'}{\Delta \uplus \{\rho\};\Gamma \vdash \mathsf{upd}^{\sigma}[\rho](e_1,\ e_2);\ e_3 : \sigma'}\,\text{t_update}$$

$$\begin{array}{rcl} E & ::= & \ldots \mid \mathsf{st}\ \theta\ \mathsf{in}\ E \mid \mathsf{mkref}^{\sigma}[\rho](E) \mid \mathsf{deref}^{\sigma}[\rho](E) \mid \\ & & \mathsf{upd}^{\sigma}[\rho](E,\ e_1);\ e_2 \mid \mathsf{upd}^{\sigma}[\rho](v,\ E);\ e \\ P & ::= & \langle \Delta,\ \mathsf{st}\ \theta\ \mathsf{in}\ e \rangle \\ A & ::= & \langle \Delta,\ \mathsf{st}\ \theta\ \mathsf{in}\ v \rangle \end{array}$$

$$\theta^{\rho} \equiv \bigcup_{\theta} \{\rho_i \cdot x_i : \sigma_i \mapsto v_i\}\ \text{where}\ \rho = \rho_i \qquad\qquad \theta^{\Delta} \equiv \bigcup_{\rho \in \Delta} \theta^{\rho}$$

...

$$\begin{array}{lrcl}
\text{r_pure} & \langle \Delta,\ \mathsf{st}\ \theta\ \mathsf{in}\ E[e] \rangle & \longmapsto & \langle \Delta,\ \mathsf{st}\ \theta\ \mathsf{in}\ E[e'] \rangle\ \text{where}\ e \overset{\text{pure}}{\longmapsto} e' \\
\text{r_halt} & \langle \Delta,\ \mathsf{st}\ \theta\ \mathsf{in}\ E[\mathsf{halt}^{\sigma}] \rangle & \longmapsto & \langle \Delta,\ \mathsf{st}\ \theta\ \mathsf{in}\ \mathsf{halt}^{\sigma} \rangle \\
\text{r_get_put} & \langle \Delta \uplus \{\rho\},\ \mathsf{st}\ \theta\ \mathsf{in}\ E[\mathsf{get}[\rho](\mathsf{put}[\rho](v))] \rangle & \longmapsto & \langle \Delta \uplus \{\rho\},\ \mathsf{st}\ \theta\ \mathsf{in}\ E[v] \rangle \\
\text{r_letr} & \langle \Delta,\ \mathsf{st}\ \theta\ \mathsf{in}\ E[\mathsf{letr}\ \rho\ \mathsf{in}\ e] \rangle & \longmapsto & \langle \Delta \uplus \{\rho\},\ \mathsf{st}\ \theta\ \mathsf{in}\ E[e] \rangle \\
\text{r_only} & \langle \Delta \uplus \Delta',\ \mathsf{st}\ \theta^{\Delta} \uplus \theta^{\Delta'}\ \mathsf{in}\ E[\mathsf{only}\ \Delta'\ \mathsf{in}\ e] \rangle & \longmapsto & \langle \Delta',\ \mathsf{st}\ \theta^{\Delta'}\ \mathsf{in}\ e \rangle \\
\text{r_dealloc} & \langle \Delta \uplus \{\rho\},\ \mathsf{st}\ \theta^{\Delta} \uplus \theta^{\rho}\ \mathsf{in}\ e \rangle & \longmapsto & \langle \Delta,\ \mathsf{st}\ \theta^{\Delta}\ \mathsf{in}\ e \rangle\ \text{where}\ \rho \notin FRV(e) \\
\text{r_merge} & \langle \Delta,\ \mathsf{st}\ \theta_1\ \mathsf{in}\ E[\mathsf{st}\ \theta_2\ \mathsf{in}\ v] \rangle & \longmapsto & \langle \Delta,\ \mathsf{st}\ \theta_1 \uplus \theta_2\ \mathsf{in}\ E[v] \rangle \\
\text{r_mkref} & \langle \Delta \uplus \{\rho\},\ \mathsf{st}\ \theta\ \mathsf{in}\ E[\mathsf{mkref}^{\sigma}[\rho](v)] \rangle & \longmapsto & \\
 & & & \langle \Delta \uplus \{\rho\},\ \mathsf{st}\ \{\rho \cdot x : \sigma \mapsto v\}\ \mathsf{in}\ E[\rho \cdot x] \rangle \\
\text{r_deref} & \langle \Delta \uplus \{\rho\},\ \mathsf{st}\ \theta' \uplus \{\rho \cdot x : \sigma \mapsto v\}\ \mathsf{in}\ E[\mathsf{deref}^{\sigma}[\rho](\rho \cdot x)] \rangle & \longmapsto & \\
 & & & \langle \Delta \uplus \{\rho\},\ \mathsf{st}\ \theta' \uplus \{\rho \cdot x : \sigma \mapsto v\}\ \mathsf{in}\ E[v] \rangle \\
\text{r_update} & \langle \Delta \uplus \{\rho\},\ \mathsf{st}\ \theta' \uplus \{\rho \cdot x : \sigma \mapsto v\}\ \mathsf{in}\ E[\mathsf{upd}^{\sigma}[\rho](\rho \cdot x,\ v');\ e] \rangle & \longmapsto & \\
 & & & \langle \Delta \uplus \{\rho\},\ \mathsf{st}\ \theta' \uplus \{\rho \cdot x : \sigma \mapsto v'\}\ \mathsf{in}\ E[e] \rangle
\end{array}$$

Figure 19: Calculus with References

...

$$\frac{\Delta \uplus \{\rho\}; \Gamma \vdash cv : \sigma \qquad \Delta \uplus \{\rho\}; \Gamma \vdash \theta \text{ st-ok}}{\Delta \uplus \{\rho\}; \Gamma \vdash \theta \uplus \rho \cdot x : \sigma \mapsto cv \text{ st-ok}}\text{st_ok_cvbind}$$

$$\begin{array}{lcl} J_1 & = & \Delta \uplus \Delta' \uplus \{\rho_1\}; \cdot \vdash cp_1 \triangleright \sigma_1 \leq \sigma_2 \\ J_2 & = & \Delta \uplus \Delta' \uplus \{\rho_1, \rho_2\}; \cdot, x_{cp} \triangleright \mathsf{ref}[\rho_1,\, \sigma_1] \leq \mathsf{ref}[\rho_2,\, \sigma_2] \vdash cp_2 \triangleright \sigma' \leq \sigma \\ J_3 & = & \Delta \uplus \Delta' \uplus \{\rho_1\}; \Gamma \vdash e_1 : \sigma' \\ J_4 & = & \Delta' \uplus \{\rho_2\} \Gamma, x : \sigma \vdash e_2 : \mathsf{Ans} \end{array}$$

$$\frac{J_1 \qquad J_2 \qquad J_3 \qquad J_4}{\Delta \uplus \Delta' \uplus \{\rho_1\}; \Gamma \vdash (\mathsf{gc_lemma}}\text{t_gc_lemma}$$
$$\mathsf{kill}\ \rho_1\ \mathsf{resurrect\ as}\ \rho_2$$
$$\mathsf{assuming}\ cp_1\ \mathsf{proves}\ x_{cp} : cf$$
$$\mathsf{let}\ x\ =\ \mathsf{coerce}[\![cp_2]\!](e_1)\ \mathsf{in}$$
$$\mathsf{only}\ \Delta'\ \mathsf{in}\ e_2) : \mathsf{Ans}$$

$$\begin{array}{rcl} E & ::= & \ldots \mid \mathsf{gc_lemma} \\ & & \quad \mathsf{kill}\ \rho_1\ \mathsf{resurrect\ as}\ \rho_2 \\ & & \quad \mathsf{assuming}\ cp_1\ \mathsf{proves}\ x_{cp} : cf \\ & & \quad \mathsf{let}\ x\ =\ \mathsf{coerce}[\![cp_2]\!](E)\ \mathsf{in} \\ & & \quad \mathsf{only}\ \Delta\ \mathsf{in}\ e \end{array}$$

r_gc_lemma_err
$$\begin{array}{l} \langle \Delta \uplus \Delta' \uplus \{\rho_1\},\ \mathsf{st}\ \theta\ \mathsf{in} \\ \quad E[\mathsf{gc_lemma} \\ \quad\quad \mathsf{kill}\ \rho_1\ \mathsf{resurrect\ as}\ \rho_2 \\ \quad\quad \mathsf{assuming}\ cp_1\ \mathsf{proves}\ x_{cp} : \mathsf{ref}[\rho_1,\, \sigma_1] \leq \mathsf{ref}[\rho_2,\, \sigma_2] \\ \quad\quad \mathsf{let}\ x\ =\ \mathsf{coerce}[\![cp_2]\!](v)\ \mathsf{in} \\ \quad\quad \mathsf{only}\ \Delta' \uplus \rho_1\ \mathsf{in}\ e] \rangle \longmapsto \langle \Delta \uplus \Delta' \uplus \{\rho_1\},\ \mathsf{st}\ \theta\ \mathsf{in}\ \mathsf{halt}^{\mathsf{Ans}} \rangle \end{array}$$

...

r_gc_lemma
$$\begin{array}{l} \langle \Delta \uplus \Delta' \uplus \{\rho_1\},\ \mathsf{st}\ \theta^{\Delta} \uplus \theta^{\Delta'} \uplus \theta^{\rho_1}\ \mathsf{in} \\ \quad E[\mathsf{gc_lemma} \\ \quad\quad \mathsf{kill}\ \rho_1\ \mathsf{resurrect\ as}\ \rho_2 \\ \quad\quad \mathsf{assuming}\ cp_1\ \mathsf{proves}\ x_{cp} : \mathsf{ref}[\rho_1,\, \sigma_1] \leq \mathsf{ref}[\rho_2,\, \sigma_2] \\ \quad\quad \mathsf{let}\ x\ =\ \mathsf{coerce}[\![cp_2]\!](v)\ \mathsf{in} \\ \quad\quad \mathsf{only}\ \Delta'\ \mathsf{in}\ e] \rangle \longmapsto \langle \Delta \uplus \Delta' \uplus \{\rho_2\},\ \mathsf{st}\ \theta^{\Delta'} \uplus \theta'\ \mathsf{in}\ e' \rangle \\ \quad \text{where} \quad cp' = cp_2[\mathsf{refl}[\mathsf{ref}[\rho_2,\, \sigma_2]]/x_{cp}] \\ \quad\quad \text{and} \quad v' = v[\rho_2/\rho_1] \\ \quad\quad \text{and} \quad e' = e[\mathsf{coerce}[\![cp']\!](v')/x] \\ \quad\quad \text{and} \quad \theta^{\rho_2} = \theta^{\rho_1}[\rho_2/\rho_1] \\ \quad\quad \text{and} \quad \theta' = (\bigcup_{b \in \theta^{\rho_2}} \mathit{newbind}(b)) \\ \quad\quad \text{and} \quad \mathit{newbind}(\rho_2 \cdot x' : \sigma_1 \mapsto cv) = \\ \quad\quad\quad\quad \rho_2 \cdot x' : \sigma_2 \mapsto \mathsf{coerce}[\![cp_1]\!](cv) \\ \quad\quad \text{and} \quad \mathit{newbind}(\rho_2 \cdot x' : \sigma' \mapsto cv) = \rho_2 \cdot x' : \sigma' \mapsto cv \end{array}$$

Figure 20: Calculus with Covariant References

Figure 20 describes the semantics in detail. We adjust our notion of a well formed store to account for the new lazy coercions. The rule in the static semantics for the gc_lemma is relatively complicated since it combines several simpler rules into one atomic term.

Judgment J_1 simply states that under the current context with no subtyping assumptions cp_1 is a valid coercion proof that demonstrates $\sigma_1 \leq \sigma_2$. Judgment J_2 establishes that $\sigma' \leq \sigma$ given a subtyping context where we may assume $\mathsf{ref}[\rho_1, \sigma_1] \leq \mathsf{ref}[\rho_2, \sigma_2]$. Judgment J_3 establishes that e_1 has a type appropriate for the coercion and judgment J_4 establishes that e_2 does not return and can be evaluated in the proper context. Notice that judgments J_2 and J_4 are the only rules that assume ρ_2 is a valid region. Also note that when typing e_2 we may not refer to ρ_1.

Because of region aliasing at runtime we cannot be assure that the regions in Δ' to do not contain aliases of ρ_1, if this occurs our system will be unsound. To correct for this problem we add a runtime check in our dynamic semantics and immediately terminate our program in this case.

If at runtime ρ_1 is not aliased to a value in Δ' we replace all occurrences of ρ_1 with ρ_2 build a new heap where we insert the appropriate coercions so that values previously of type σ_1 are lazily coerced into values of type σ_2. Again all these operations are notational and would have no real operational effect in practice.