Type-Preserving Garbage Collectors

Daniel C. Wang Andrew W. Appel

Department of Computer Science
Princeton University
Princeton, NJ 08544 USA

Abstract

By combining existing type systems with 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. Our type safety guarantee is non-trivial since not only does it guarantee the type safety of the garbage collector, but it guarantees that the collector preservers the type safety of the program being garbage collected. 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

We outline an approach, based on ideas from existing type systems, to build a type-preserving garbage collector. We can guarantee that the collector preserves the types of the mutator's data-structures. Traditionally a collector is primitive runtime service outside the model of the programming language, the type safety of running programs depends on the assumption that the collector does not violate any typing invariants. However, no realistic system provides a proof of this assumption. Our primary contribution is to demonstrate how to construct tracing garbage collectors so that one can formally and mechanically verify, through static type checking, that the collector does not violate any typing invariants of the mutator.

Our approach is simple: make the collector a well typed function written in the same typed intermediate language used by the compiler of the mutator's source language.

This research was supported in part by DARPA award F30602-99-1-0519 and by the IBM University Partnership Program.
To appear in the *28th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL '01)*, January 2001.

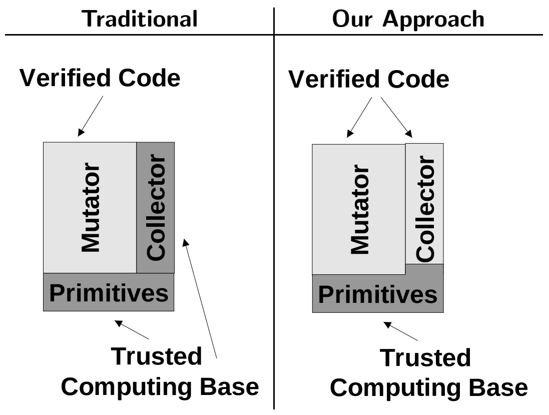

Figure 1: Reduced Trusted Computing Base

Garbage collection is no longer a primitive runtime service, uses no unsafe primitives, and is part of our model of the programming language. Since the collector and mutator are both well typed, we appeal to the fact that "Well typed programs do not go wrong." Our language uses a region based type system [27] for safe primitive memory management. The collector is built on top of these safe region primitives. Regions are used to implement the semi-spaces of a traditional copying collector. The region type system allows us to verify that it is safe for the collector to deallocate a semi-space that contains only garbage.

Comparison to region inference. Our collector dynamically traces values at runtime, allowing for more fine-grain and efficient memory management than systems that use region inference, which may take asymptotically more space than a simple tracing garbage collector. From a different perspective, our collector is merely a particular way of writing programs in a language that 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 [26]. Our approach suggests how to cleanly integrate compile-time memory management techniques with traditional runtime techniques to gain the benefits of both approaches. We consider this to be an important secondary contribution.

Comparison to proof-carrying code. Safety architectures such as Java byte-code verification and proof-carrying code statically verify safety properties of code provided by an untrusted code producer [21, 13]. These systems rely on a trusted garbage collector to safely handle memory deallocation. Our approach allows us to verify the safety of the mutator and collector, placing the collector outside of the trusted computing base (TCB). Our type-preserving collector relies on a few new low-level runtime primitives, but the total size of the TCB is smaller[1] (see Figure 1). Since our TCB is smaller we are able to provide a stronger guarantee of safety. Although we verify programs through static type checking, existing proof-carrying code systems can adapt our techniques to reduce the TCB in the same way.

Even if we are willing to trust that a particular garbage collector is correctly implemented, formalizing the invariants needed to properly interface a mutator with the collector will complicate the safety policy in a proof-carrying code system. Also we must trust that the more complex safety policy is sufficient to guarantee safety. Even *conservative garbage collectors*, which have simpler interfaces by conservatively inferring needed type information at runtime, require the compiler to preserve subtle invariants [8].

Formal treatment of collector interfaces. Another important contribution of our work is the ability to think about garbage collector interfaces in a statically checkable way. We can check that the mutator uses the interface properly, and more importantly that the interface is sufficient for the collector to preserve the type safety of the mutator. Many of the bit-level details of garbage collector interfaces can be described in a high-level and type-safe way, using simple and standard typing constructs. In particular we describe one way to implement "stack walking" [11] without an explicit table that maps the return address of a function to a stack frame layout. We are able to do this by encoding the table implicitly and in a checkable way.

Statically catching these bugs makes the system more secure, easier to debug, more flexible, and potentially more efficient. We can catch interface bugs, such as the failure to include a live value in the root set or providing incorrect type information, at compile time. Since the collector is not a fixed trusted piece of the system, individual programs can provide a specialized collector which may improve program performance.

A traditional copying collector. Figure 2 illustrates 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 respectively. 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 and the *live roots* which point to 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 so, the collector creates a fresh allocation space (`to`) then makes a deep copy of the live roots into the *to-space*. All the data reachable from the new roots (`roots'`) should live in the to-space. The collector can now 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.

In order to guarantee that the from-space can be safely deallocated, we must be certain that "the rest of the program" never accesses values allocated in the from-space. If our program is written in continuation passing style, we can easily enforce this invariant by assigning a static type to `k` so that it cannot access values in the from-space. We can easily formalize this intuition into a relatively simple type system.

Technical challenges. Building a type-preserving collector does not rely on a single key technical advance, but results from the combination of several advances in typed compilation. The key issues that need to be addressed are:

1. Copying
2. Source language abstractions
3. Deallocation
4. Pointer sharing

If the static type of every object is known at compile time, it is easy to write a well typed function that produces a copy of the object with the same type. However, when the type is not known at compile time, because of polymorphism or issues of separate compilation, this task becomes more challenging. Fortunately work in the area of intensional type analysis [14, 10] and other forms of ad-hoc polymorphism that use dictionary passing [30] provide clean solutions to this problem.

Traditional collectors violate data-abstraction guarantees that are present in the source language. The "private" fields of an object in Java or "private" environment of a closure in ML cannot remain private to the garbage collector. We must decided if we wish to preserve these abstraction guarantees or violate data-abstraction when performing garbage collection.

For example there are several well known techniques for type-preserving closure conversion. [16, 20, 28] Many of the schemes provide strong guarantees that they preserve source level abstractions. In practice many compilers still must provide extra type information that describes the layout of "abstract" objects for the garbage collector, so claims of abstraction preservation break down at the level of the garbage collector. Other closure conversion techniques for first-order target languages [28] provide much weaker abstraction-preservation guarantees and make the layout of closures explicit during translation. Intensional type analysis formalizes the passing of extra type information (typically provided by the compiler for the garbage collector) in a fully type-safe way [10]. We touch on some of the tradeoffs of these approaches in Section 2.

Collectors must use some primitive memory management service to allocate and deallocate the from-space and the to-space. We must verify that the 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

[1]The primitives in our prototype system are implemented in approximately 200 lines of C code while a realistic garbage collector is in the range of 3000 lines of C.

```
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, roots')
 else k(from, roots)
```

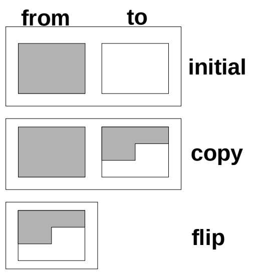

Figure 2: Traditional Garbage Collector

memory management [27, 1, 9, 7]. We can use the memory management primitives provided by a region system to guarantee that it is safe to deallocate the from-space after the garbage collector has copied all the live data into the to-space.

Pointer sharing is preserved 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. Or approach requires some inelegant ad-hoc reasoning, but our technique is as efficient as current unsafe techniques and can be formally proven sound.

In Section 2 we informally describe the language we will use to build our type-preserving collector. 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 Type-Preserving Garbage Collectors

Each technical challenge can be solved with several different techniques. To simplify our presentation we will choose the simplest solution for each challenge, and discuss more complex alternatives. We only consider a first-order language where all types are known at compile time. 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 [17, 23]. Under these assumptions we can generate a copy function for each type of object. We could avoid the need for a first-order compilation approach and also support separate compilation better if we used the technically more sophisticated techniques of intensional type analysis. 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.

A Simple Region Type System. The first-order assumption simplifies the region system by allowing us to ignore latent effects. For our purposes the region type system need not be particularly advanced. We do not need to separate read and write effects, support effect polymorphism, account for latent effects (since our language is first order), or allow for dangling pointers. All of these features are included in the original Tofte-Talpin region calculus [27].

However, one feature that our region calculus must support, but is not provided by the original Tofte-Taplin system, is early deallocation. The region system of Crary, Walker, et al [9] supports early deallocation. It is sufficient for our purposes but is still more complex then needed because of their static approach to handling issues of region aliasing. A simple region type system suitable for our purposes that supports early deallocation and handles region aliasing through a simple runtime check is described in [32]. We will use this system in the description of our work, because of its simplicity.

Violations of Abstraction. The first-order restriction forces us to turn higher-order objects such as closures, which are normally abstract, into concrete values with an explicit representation. This has the advantage that our collector can now check if two closures have the same "pointer address," and perform other operations that would typically violate abstractions in higher-order languages.

While these abstraction violations are troubling, they merely reflect the fact that existing garbage collection techniques tend to violate abstraction. However, we believe this violation of abstraction is not fundamental and that one can easily develop techniques that are not only type-preserving but also-abstraction preserving. However, it is not clear if these abstraction preserving techniques are as efficient as the current known techniques that violate abstraction, and efficiency is an important concern when developing garbage collectors.

In the next section, 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 explicitly typed first-order ML-like language with regions and early deallocation. The type system is not particularly novel so we will discuss it only informally.

3 Example: `itrev`

Source program. Figure 3 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 allocate a new stack frame for this call. Note when the program first calls `itrev` the call is not a tail call, so we must allocate a trivial stack frame for this call. As the function recursively descends `l` the previous list cells, contained in the dotted box in the figure, are garbage

```
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) (* non-tail call *)
in rl
```

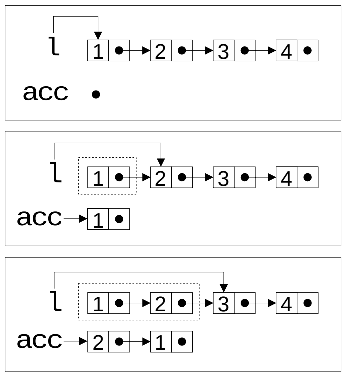

Figure 3: Iterative List Reverse

and can be reclaimed. The function therefore, need only retain a constant amount of live data in addition to the list itself. This simple reasoning cannot be applied in systems that use region inference to manage memory.

Region inference would not allow us to immediately free each list cell in `l` after we have traversed it. A region system would force us to hold onto all the cells of `l` until the function returns `acc`. 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 [4, 6, 24, 31].

We will convert the program in Figure 3 into an equivalent program that includes a function to garbage-collect dead values and is still well typed. 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. If we CPS convert our source program, reasoning about the control flow of the program becomes easier. However, since our language is first-order we cannot use a standard CPS conversion algorithm, which requires higher-order functions. Instead we adapt a first-order closure conversion technique outlined by Tolmach with a standard CPS conversion. Figure 4 illustrates Tolmach's closure conversion technique. Notice that the types of any free variables are captured in the type of the closure [28].

Figure 5 is the result of applying these both the CPS and closure conversion transformations on our example. Notice the new type `cont` which is the type of return continuations for the function `itrev`. All functions have a return type of Ans, which means they do not return. This type contains one data constructor `Ret_rl` which is needed for our one non-tail call in the original program. In general each call site of `itrev` will require one new data constructor to represent each distinct return continuation. Also note that we implicitly assume we have access to the whole program at this point.

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 [11]. 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. So we can replace a low-level table of bitmaps with a set of high-level type declarations.

The chief disadvantage of first-order closure conversion is that it makes separate compilation more difficult.[2] However, providing true separate compilation using standard higher-order techniques that preserve abstraction and have better separate compilations properties is not as simple as it may seem. Even these techniques must have a method of merging type information at link time or force all objects to be uniformly tagged, which is often undesirable.

Region annotated. We have been informally arguing about where and when objects are allocated. Figure 6 shows our program with explicit region annotations. Notice that the type `lst` in figure 5 becomes a type constructor `lst[`ρ`]` 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 `Cons(int, lst[`ρ`]) at` ρ.

Both the `itrev` and `apply` functions each take a single region parameter (ρ_{alloc}), which corresponds to the allocation pointer in a normal untyped system. When we allocate a new list cell we use the notation `lst[`ρ_{heap}`].Cons(1,...)` which instantiate the region parameter (ρ) of the type constructor `lst` to ρ_{heap} and indicates that the new list cell will be allocated in the region ρ_{heap}. We have assigned regions to types so that values are allocated in one global region, which acts like a traditional heap. When we call `itrev` we instantiate its region parameter ρ_{alloc} to ρ_{heap}. We could apply a more refined region local analysis to avoid heap-allocating an object when the lifetime of the object is locally 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 first class continuations as well as simplifying the reasoning of safety. However, heap-allocating return continuations could impact performance in an undesirable way. A system extended with linear types, along with a

[2]Tolmach outlines a separate compilation technique that requires special support from the linker.

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

Figure 4: 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 (* bind return value rl *)
   in rl ; halt() (* exit program *)

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

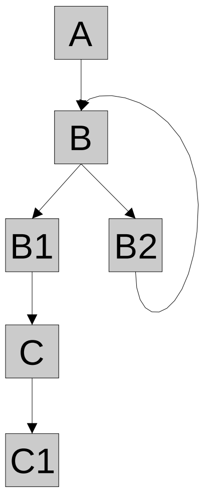

Figure 5: CPS-Converted and Closure-Converted Program

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

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

letr ρ_heap in (* initial program heap *)
let l = lst[ρ_heap].Cons(1, ...) in (* heap allocate list *)
let k = cont[ρ_heap].Ret_rl  (* create return continuation *)
in itrev[ρ_heap](k, l, lst[ρ_heap].Nil)
```

Figure 6: Program `itrev` after Region Annotation

set of simple syntactic restriction would allow us to stack allocate return continuations.

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 [11]. It requires that the compiler 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 7 shows such a "safe point" in our program. Depending on some heuristic the code either continues executing or packages 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, since not including a root would result in a scoping error or a violation of the region type system. More importantly, we would be able to easily where the error was by examining the code statically, 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 the only term. Figure 8 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 dynamically bound to ρ_{to}.

In general the only expression takes an arbitrary set of region variables. At runtime we simply note what regions are dynamically bound to the region variables passed to the only expression and safely deallocate any other regions, since they are not needed to evaluate the rest of the program. The cost of this deallocation operation is at worst linearly related to the number of live regions. Our safe garbage collector needs at most two live regions at any time, so in practice the cost of this dynamic approach is negligible. This dynamic approach to early deallocation of regions is a novel approach which is simpler than current static approaches to early deallocation and more expressive.

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))
```

The expression put$[\rho](1)$ stores the integer into the region ρ_1 and returns a reference to the integer. The term get$[\rho_b](x)$ reads an integer from the region ρ_b. Notice that if the program executes the first branch of the conditional then `f` behaves as expected. However, if we execute the second branch then at runtime the region variables ρ_a and ρ_b are both bound to the same region 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 region variables through various typing disciplines [9]. The static approaches do not incur any runtime overhead, but are relatively complex systems and would disallow us from writing the program above.

Using our dynamic approach we write `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 [12]. 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 9 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 Deutsch-Schorr-Waite pointer reversal algorithm [22, 29, 25, 31]. Note that the function `copy_cont` performs an operation equivalent to "walking the stack". Since we have CPS converted our program the continuation, `k`, represents the current stack frame. It may be the case that we can adapt the higher-order techniques to provide true abstraction and separate compilation in the presence of a garbage collector by requiring each abstract object to provide a method[3] to copy or trace the object. It is not clear what the software engineering and performance issues are

[3] A closure can be thought of a an object with a single "apply" method.

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

Figure 7: 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_alloc](...):Ans = ... and apply[ρ_alloc](...):Ans = ...
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 (* deallocate ρ_from *)
    case roots' of
     Ret_itrev(k, l, acc) ⇒ itrev[ρ_to](k, l, acc)
and copy_gc_cont[ρ_from, ρ_to](x:gc_cont[ρ_from]):gc_cont[ρ_to] = ...
...
```

Figure 8: "Flipping" from and to space

```
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_alloc](...):Ans = ... and apply[ρ_alloc](...):Ans = ...
and gc[ρ_from](...):Ans = ...
and copy_gc_cont[ρ_from, ρ_to](x:gc_cont[ρ_from]):gc_cont[ρ_to] =
 case x of Ret_itrev(k, l, acc) ⇒
  let k' = copy_cont[ρ_from, ρ_to](k) in (* walk the "stack" *)
  let l' = copy_lst[ρ_from, ρ_to](l) in
  let acc' = copy_lst[ρ_from, ρ_to](acc)
  in gc_cont[ρ_to].Ret_itrev(k', l', acc')
and copy_lst[ρ_from, ρ_to](x:lst[ρ_from):lst[ρ_to] = ...
and copy_cont[ρ_from, ρ_to](x:cont[ρ_from]):cont[ρ_to] = ...
...
```

Figure 9: Copying roots

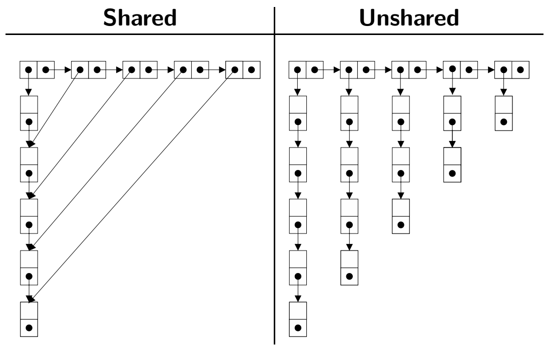

Figure 10: Shared vs. Unshared values

for this technique so we consider it to be future work. A more serious problem with our copy function is that it does not preserve pointer sharing.

Preserving Sharing. Consider the datastructure in the first half of Figure 10. 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 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 11 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 12 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 mutable 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. 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[4] 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 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 value of type `lst`, if after casting it 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, 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 nonstandard and ad-hoc form of subtyping on references. This 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 of Crary, Walker,

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

```
prim objId : [$\alpha$] $\alpha$ $\rightarrow$ int
tycon tbl :: Rgn $\rightarrow$ Typ $\rightarrow$ Typ $\rightarrow$ Typ = ...
fun newTbl[$\rho_{tbl}$, $\alpha$, $\beta$](sz:int):tbl[$\rho_{tbl}$, $\alpha$, $\beta$] = ...
fun inTbl[$\rho_{tbl}$, $\alpha$, $\beta$](t:tbl[$\rho_{tbl}$, $\alpha$, $\beta$], key:$\alpha$):bool = ...
fun getTbl[$\rho_{tbl}$, $\alpha$, $\beta$](t:tbl[$\rho_{tbl}$, $\alpha$, $\beta$], key:$\alpha$):$\beta$ = ...
fun addTbl[$\rho_{tbl}$, $\alpha$, $\beta$](t:tbl[$\rho_{tbl}$, $\alpha$, $\beta$], key:$\alpha$, val:$\beta$):unit = ...

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

fun itrev[$\rho_{alloc}$](...):Ans = ...
...
and share_copy_lst[$\rho_{tbl}$, $\rho_{from}$, $\rho_{to}$]
  (t:tbl[$\rho_{tbl}$, lst[$\rho_{from}$], lst[$\rho_{to}$]],x:lst[$\rho_{from}$]):lst[$\rho_{to}$] =
 case x of Nil $\Rightarrow$ lst[$\rho_{to}$].Nil
 | Cons(hd, tl) $\Rightarrow$
    if inTbl[$\rho_{tbl}$, lst[$\rho_{from}$], lst[$\rho_{to}$]](x) then (* is forwarded? *)
     getTbl[$\rho_{tbl}$, lst[$\rho_{from}$], lst[$\rho_{to}$]](x)
     else let hd' = hd in
      let tl' = share_copy_lst[$\rho_{tbl}$, $\rho_{from}$, $\rho_{to}$](t,tl) in
      let x' = lst[$\rho_{to}$].Cons(hd',tl')
      in addTbl[$\rho_{tbl}$, lst[$\rho_{from}$], lst[$\rho_{to}$]](x,x') ; (* set forwaded *)
         x'
...
```

Figure 11: Preserving Sharing with a Hash-Table

```
tycon ref :: Rgn $\rightarrow$ Typ $\rightarrow$ Typ
type gc_lst[$\rho_{from}$, $\rho_{to}$] =  Nil
 | Cons(ref[$\rho_{from}$,fwd_ptr[$\rho_{to}$]], int, gc_lst[$\rho_{from}$, $\rho_{to}$]) at $\rho_{from}$
and fwd_ptr[$\rho_{to}$] = NULL | PTR(lst[$\rho_{to}$])
and lst[$\rho_{to}$] = Nil
 | Cons(ref[$\rho_{to}$, fwd_null], int, lst[$\rho_{to}$]) at $\rho_{to}$
and fwd_null = NULL
fun itrev[$\rho_{alloc}$](...):Ans = ...
...
and share_copy_lst[$\rho_{from}$, $\rho_{to}$](x:gc_lst[$\rho_{from}$, $\rho_{to}$]):lst[$\rho_{to}$] =
 case x of Nil $\Rightarrow$ lst[$\rho_{to}$].Nil
  | Cons(f, hd, tl) $\Rightarrow$
    (case deref[$\rho_{from}$](f) of NULL $\Rightarrow$
      let hd' = hd in
      let tl' = share_copy_lst[$\rho_{from}$, $\rho_{to}$] in
      let l = lst[$\rho_{to}$].Cons(mkref[$\rho_{to}$](fwd_null.NULL), hd' , tl')
      in f := l; l
     PTR(l) $\Rightarrow$ l)
```

Figure 12: Encoding Forwading Pointers

et al [9] 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. See [32] which sketches the soundness of the approach for a simpler core calculus. It is important to note that we still must at run time check that ρ is not aliased by any other region variable, so that the new region variable ρ' refers to a unique region. This extra alias check is need for this approach to be completely sound, but all our alias checks would be unnecessary in the system of Crary, Walker, et al.

5 Preliminary Performance Evaluation

The approach we have outlined is asymptotically competitive with existing garbage collection algorithms. However we cannot neglect constant factors and other important pragmatic issues, if we wish to build a practical system. One issue 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 [11] 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 these techniques are not sufficient we can adopt the techniques such as intensional type analysis [14] to avoid having a distinct copy function for every unique type. For our preliminary evaluation we will ignore the issues of code size and just examine efficiency of the currently described system.

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

itrev Iterative list reverse (n = 10,000)

appel1 Program designed to demonstrate issues of space complexity (n = 1000) [27]

inline Inline variant of appel1 (n = 1000) [27]

appel2 Program designed to demonstrate issues of space complexity (n = 1000) [27]

ackermann Ackermann's function evaluated (n = 3, m = 6) [27]

fib Recursive Fibonacci (n = 33)

hsum Sum the value in a heap allocated list (n = 1000) [27]

quicksort Quicksort randomly generated list (n = 1000) [27]

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

sum Recursive sum of the first n integers (n = 1000) [27]

These programs are not a representative workload. However, they are sufficient for a 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.

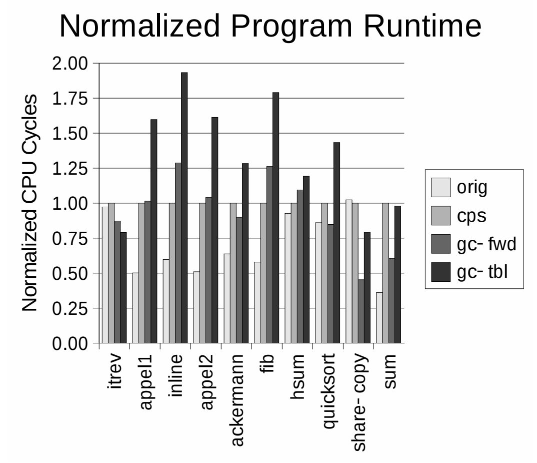

Figure 13: Relative Runtime Performance

Compiler. We have modified the back end of the `MLton` [17] to accept source programs that include a safe garbage collector. The `MLton` compiler emits C code that is then processed 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.

For each source programs we collect data for the following variants:

orig Original program passed directly to `MLton`

cps Program run through CPS transform and first-order closure conversion, run with `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 hash table to preserve sharing

To better understand the impact of CPS conversion, we measure the runtime of programs using the unsafe collector before (**orig**) and after CPS conversion (**cps**). We finally measure the performance of two different safe collectors, which differ only in their approach to sharing preservation; one uses forwarding pointers (**gc-fwd**), the other a hashtable (**gc-tbl**).

In a production system we would synthesize a safe collector after high-level optimizations, but because of the structure of `MLton` it was more convenient to synthesize a collector before many high-level optimizations. 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 system C compiler (`gcc`).

Effect of CPS Conversion. Figure 13 shows the total wall-clock run time for each program and variant normalized by the performance of the unoptimized CPS-converted program. Immediately, one can see that the CPS conversion

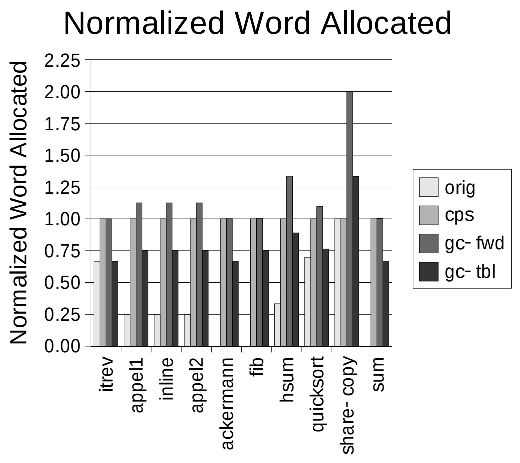

Figure 14: Relative Number of Words Allocated

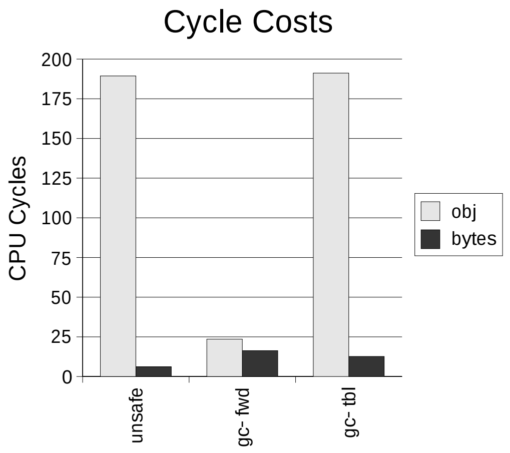

Figure 16: Per-word and per-object cycle costs

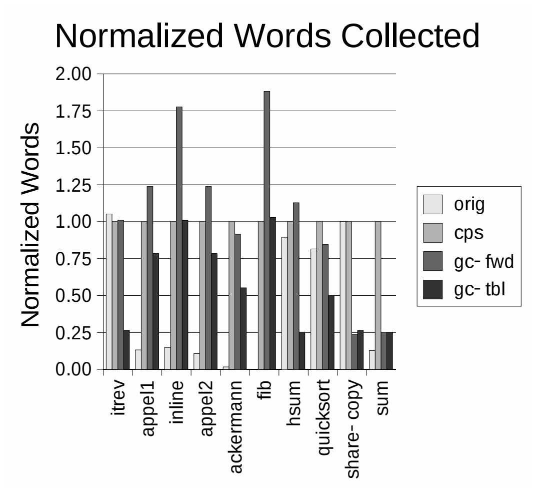

Figure 15: Relative Number of Words Garbage Collected

can cause more than a factor of two performance degradation when compared to the original program, which is stack allocating activation records. We are using a simple flat-closure representation; more advanced closure representation techniques can significantly reduce the amount of allocation.[2, 3] Figure 14 shows that our CPS converted programs are allocating significantly more heap data[5], which accounts for the performance difference. Also note that programs using a safe collector with a hash table 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 an 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 hash table is not incurring this extra space overhead for tagging or forwarding, but uses more auxiliary space during garbage collection.[6]

[5]Notice that some programs did not allocate any heap data originally.

[6]The extra auxiliary space need for garbage collection is not accounted for in the figure.

Unfortunately, synthesizing a garbage collector before optimizing prevents 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. `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 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 even when the unsafe version of the program is stack-allocating activation records. This naive comparison is misleading, because the various programs allocate different amounts of data at different times. Because each program's allocation behavior is different, the number of words actually garbage-collected varies. Figure 15 shows the relative amount of data actually garbage-collected for each program.

This explains why in the case of `itrev` our safe collector, which uses a more costly hash table to preserve 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 hash table to preserve sharing, our collector will be invoked less frequently.[7] In this case the program using the safe collector with a hash table 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 Measurements. With the caveat that raw copying performance is not an accurate measure of the performance impact of a garbage collection scheme, we report the raw copying performance of our collector, by assuming

[7]In the case of `share-copy`, which is allocating more data, because of our dynamic heap resizing policy it is being invoked at different times when there is less live data to be collected.

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 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. We omit numbers for the **orig** case since it is using exactly the same unsafe collector as **cps**. 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. We must add a caveat that with such small programs we are ignoring important caching effects in our analysis.

Our experiments suggest that if we modify our framework so that we can stack allocate return continuations, and if caching-related effects can be addressed our safe collector should be competitive with traditional unsafe techniques.

6 Conclusions and Future Work

Although our approach as presented is not practical for general-purpose systems, we believe practical systems can be built by extending our current work. The most important insights are that a general-purpose collector can be built on top of a set of much simpler primitives, and that when standard type systems are too weak, we can rely on runtime checking or simply add "the right lemma" and encode what amounts to a small proof sublanguage to establish important preconditions needed for any ad-hoc reasoning that does not fit into a standard framework.

At a high-level, garbage collection algorithms move objects from one abstract set to another. Particular garbage collection algorithms differ in how these abstract sets of objects are implemented. In our type-preserving collector each abstract set of objects corresponds to a region. Our technique is not dependent on any particular implementation of the region primitives.

In the past region have been implemented as contiguous allocation arenas. If we implement regions as doubly-linked list of objects rather than contiguous allocation arenas, we can build a "fake copying" collector [33]. The "fake copying" scheme forms the basis for incremental techniques such as Bakers's Treadmill [5]. We maybe be able to use this observation as the basis for building safe incremental collectors. The mark bits used in mark-sweep and mark-compact collectors can also be seen as a simple set membership bit. We believe that with an appropriate implementation of the underlying region primitives, mark-sweep and mark-compact collection schemes could be implemented.

We would like to investigate how to integrate purely static memory management techniques [24, 31] with our system. [18] takes our basic approach and extends it to use the more sophisticated techniques of intensional type analysis, and outlines an approach for encoding a generational collector as well as presenting an alternative approach to forwading pointers.

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 [19] 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 the underlying implementation details. Our semantics is sufficiently detailed 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 of our garbage collection algorithm by simply relying on type soundness.

Ideally we would like to have a spectrum of static and dynamic memory management techniques so one can mix techniques in a clean, efficient, and safe way. We would like to investigate in more detail the abstraction related issues we have mentioned. Although our technique is type-preserving it is still not abstraction-preserving. We believe research in this direction may lead to more modular memory management techniques.

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] Andrew W. Appel and Zhong Shao. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, January 1996.

[3] Andrew W. Appel and Zhong Shao. Efficent and safe-for-space closure conversion. *ACM Transactions on Programming Languages and Systems*, 22(1):129–161, January 2000.

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

[5] Henry G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

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

[7] 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.

[8] 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.

[9] 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.

[10] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In ICFP [15], pages 301–312.

[11] 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.

[12] 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.

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

[14] 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 130–141. ACM Press, January 1995.

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

[16] 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.

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

[18] Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. Technical Report Yale/DCS/1205, Yale University, 2000.

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

[20] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Conference Record of the Twenty-fifth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 85–97. ACM Press, 1998.

[21] 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.

[22] 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.

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

[24] 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.

[25] Jonathan Sobel and Daniel P. Friedman. Recycling continuations. In ICFP [15], pages 251–270.

[26] 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.

[27] 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.

[28] 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.

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

[30] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 60–76. ACM Press, January 1989.

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

[32] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors (extend version). Technical Report TR-624-00, Princeton University, 2000.

[33] Thomas Wang. The MM garbage collector for C++. Master's thesis, California State Polytechnic University, October 1989.