Safe Garbage Collection = Regions + Intensional Type Analysis

Daniel C. Wang Andrew W. Appel

Department of Computer Science
Princeton University
Princeton, NJ 08544 USA

Tech Report TR-609-99

October 19, 1999

Abstract

We present a technique to implement type-safe garbage collectors by combining existing type systems used for compiling type-safe languages. We adapt the type systems used in *region inference* [16] and *intensional type analysis* [8] to construct a safe stop-and-copy garbage collector for higher-order polymorphic languages. Rather than using region inference as the primary method of storage management, we show how it can be used to implement a garbage collector which is provably safe. We also introduce a new region calculus with non-nested object lifetimes which is significantly simpler than previous calculi. Our approach also formalizes more of the interface between garbage collectors and code generators. The efficiency of our safe collectors are algorithmically competitive with unsafe collectors.

1 Introduction

We present a technique to implement type-safe garbage collectors by combining existing type systems used for compiling type-safe languages. We adapt the type systems used in *region inference* [16] and *intensional type analysis* [8] to construct a safe stop and copy garbage collector for higher-order polymorphic languages. Our approach has all the benefits of traditional tracing garbage collection as well as the benefits that come with type safety.

Tracing garbage collectors perform two potentially unsafe operations:

- Traverse arbitrary runtime values to identify live objects.
- Explicitly allocate and deallocate memory.

Tracing an arbitrary value is similar to other type-dependent functions such as pretty-printers and polymorphic equality functions. There are several efficient techniques for implementing type-dependent functions that preserve type safety[14, 13].

There are also several well studied type systems that allow one to verify the safety of explicit memory management in higher-order polymorphic languages[16, 1, 4]. Often these systems are represented as alternatives to traditional garbage collection techniques.

Contributions. We show how to use a simple type system to implement a provably safe garbage collector. The collector is safe in the sense that a correct program will not "go wrong" after the garbage collector runs. This lets us build systems whose memory utilization is as good as existing safe systems, but whose safety does not rely on the correct implementation of a trusted tracing garbage collector.

We adapt the ideas and type systems from the work on regions[16, 1, 4]. The basic idea is that a mutator always allocates in a fixed region. When the collector is invoked, it copies all the live data from one region into a newly allocated region. The regions act as the semi-spaces of a stop-and-copy collector. We use intensional type analysis to assign a type to our copy function that guarantees our collector has performed a deep copy of the live data. The older region can then be safely deallocated. This approach requires regions with non-nested life-times.

We introduce a new region calculus, λ_ρ, with non-nested object life-times. Our new calculus is simpler and more expressive than existing calculi with non-nested life-times, because we are willing to perform checks at runtime when deallocating regions. Furthermore, we describe how to use several existing type systems and type based compilation techniques so we can apply our safe garbage collection approach to higher-order polymorphic languages. Our approach also formalizes much more of

the interface between garbage collectors and code generators.

Correctness vs. Safety. Rather than attempt to guarantee correctness, we are interested in providing the following safety guarantees:

- No live values are reclaimed prematurely.
- The types of objects are preserved by the collector.

After the garbage collector is run, we assure that the program will not "go wrong" because of type errors or because memory was reclaimed prematurely. Most correctness proofs of garbage collection guarantee that live values before the collection are somehow isomorphic to the live values after collection[6].

Requirements. It is easy to implement a safe garbage collector by simply implementing a collector that never reclaims memory. It is obvious that a realistic system must have other desirable properties. In particular we are interested in a safe automatic memory management system with a small trusted computing base (TCB), that allows separate compilation, and with performance and memory utilization as good as existing unsafe systems.

	static regions	tracing gc	our approach
memory utilization	?	+	+
separate compilation	–	+	+
small TCB	+	–	+
real time properties	+	?	?

Table 1: Summary of Tradeoffs

Regions vs. Tracing GC. Because the work on regions is quite new the efficiency and scalability of region inference has not been sufficiently demonstrated. Traditional tracing garbage collectors have existed for several decades and their efficiency and scalability properties are well understood. Intuitively tracing garbage collectors have more information since they are able to examine program state at runtime. The biggest advantage to the static approach, taken by region inference, is that the trusted primitives are much simpler, and therefore easier to implement correctly. Systems that rely on tracing collectors take the entire garbage collector to be a trusted "primitive".

All static approaches require some sort of global analysis to get good memory utilization, making separate compilation difficult. However, because the underlying runtime system's primitives are simple, they have good real-time properties, since each primitive is a constant-time operation. Table 1 summarizes the tradeoffs between these two approaches in comparison to ours.

Problems Using Current Regions Systems. The standard Tofte-Talpin region system [16, 3] is not sufficiently expressive to implement collectors directly. In particular the Tofte-Talpin requirement that all region lifetimes be nested in a stack-like way interferes with standard tail-call optimizations. Tofte-Talpin extend their system with a static storage-mode analysis to allow for something similar to standard tail-call optimizations[3]. However, storage-mode analysis is not expressive enough to implement a general-purpose garbage collector.

Outline. In the next section we formally describe a novel region calculus and discuss how to use it to implement what amounts to a trivial garbage collected system and we compare our system to other variants of the Tofte-Talpin region systems.

The remaining sections describe issues of scanning values, how to deal with closures and how to preserver pointer sharing. Appendix A sketches a proof of soundness for our region calculus.

2 The λ_ρ Calculus

$$
\begin{array}{rcl}
s,t & ::= & \texttt{int} \mid s \xrightarrow{\Delta} t \mid \forall\rho.s \mid (s\ \texttt{at}\ \rho) \mid \texttt{Ans} \\
\Delta & ::= & \{\} \mid \{\rho_1,\ldots,\rho_n\} \\
e & ::= & x \mid n \mid (+\texttt{1}\ e) \mid (-\texttt{1}\ e) \mid (\texttt{if0}\ e\ e_1\ e_2) \mid \\
 & & (\lambda x:s.e) \mid (e_1\ e_2) \mid (\Lambda\rho.e) \mid (e\ [\rho]) \mid \\
 & & (\texttt{letr}\ \rho\ \texttt{in}\ e) \mid (\texttt{only}\ \Delta\ e) \mid \\
 & & (\texttt{put}[\rho]\ e) \mid (\texttt{get}[\rho]\ e) \mid \\
 & & (\mu\ f:s.e) \mid (\texttt{exit}\ e)
\end{array}
$$

Figure 1: Abstract Syntax for λ_ρ

We begin with an informal account of λ_ρ, a region calculus based on the translation of a Tofte-Talpin region system into an simple variant of the polymorphic lambda calculus described by Banerjee et al. [2]. Figure 2 contains the abstract syntax of λ_ρ.

The types of λ_ρ are very similar to the type system of a standard polymorphic typed lambda calculus. The calculus above allows for polymoprhism only over region variables ρ. `Ans` is the return type of continuations. Function types are annotated with an effect set (Δ), which describes

the set of region variables that a function of that type may access. Values of type $(t\ \texttt{at}\ \rho)$ represent values allocated in region ρ.

The terms of λ_ρ are the terms of a simple polymorphic lambda calculus, where we replace type abstraction with region abstraction. A term of the form $(\texttt{put}[\rho]\ e)$ takes a region variable and an expression and allocates/boxes the value of the expression in the given region resulting in a value of type $(t\ \texttt{at}\ \rho)$ where t is the type of the expression, e. The $(\texttt{get}[\rho]\ e)$ takes a boxed value of type $(t\ \texttt{at}\ \rho)$ and unboxes the value to a value of type t.

The region variables in the `put` and `get` terms act as capabilities. Our type system guarantees that the regions used by these terms will be live. The $(\texttt{letr}\ \rho\ \texttt{in}\ e)$ term introduces and allocates a new region and evaluates its body with a fresh region bound to the region variable ρ.

Early Deallocation. The $(\texttt{only}\ \Delta\ e)$ term is our extension to a standard region calculus to support non-nested life-times. It acts as a static assertion that its body can be evaluated using the set of regions bound to the region variables in Δ, which is a subset of the region variables currently in scope. It does not introduce any new region variables. In order to safely deallocate regions it must be the case that we do not return from the evaluation of e. Our typing rules enforces this property by requiring e have the type `Ans`.

At runtime implementations will dynamically mark regions mentioned in Δ as live and reclaim unmarked regions. The set of regions, Δ, acts as the root set of a very simple garbage collector for regions. The garbage collector for regions is significantly simpler than a normal garbage collector since it is reclaiming whole regions. Regions do not need to be scanned for references to other objects, since each region can be though of as an atomic "pointer free" object.

Rather than just deallocating the most recent region on our stack of regions before the call, the `only` expression implicitly deallocates any region which is not needed for the future computation. Our calculus organizes the set of live regions not as a stack, but as a set of regions which allows for arbitrary allocation/deallocation policies.

Our type system simply keeps track of which regions need to be syntactically live for a given expression to successfully complete. The cost of `only` is linearly proportional to the number of region variables statically in scope. In a typical system the number of such variables should be a very small bounded number. Since deallocation occurs at garbage collection points, which are infrequent, the extra cost of `only` should not effect overall performance.

2.1 Operational Semantics of λ_ρ

The formal operational semantics are described by Figure 2.1 in the style of Wright and Felleisen [19] and is similar in spirit to the more detailed operational semantics of Morrisett, Felleisen and Harper [10]. We first identify a syntactic subset of terms that denote ground values, which includes values that reside in a region $(\texttt{put}[\rho]\ v)$. Evaluation contexts, E, are a subset of expressions with "holes". $E[e]$ represents an evaluation context with the hole, $[\,]$, replaced by e. Notice that the set of evaluation contexts with the holes filled by an expression is equivalent to the set of expressions. A set of region variables paired with an expression is a program, P.

Our region context acts very much like the "capability context" of Crary et al.[4]. We will assume that bound region variables are uniquely named and that substitution preserves this property. Because of this assumption we can blur the distinction between a region variable and the dynamic region value itself, since every region variable is unique.

The reduction rules rewrite programs to programs. All are standard except for the rules that manipulate the region context. Rather than model a heap directly our operational model abstracts away the details of an explicit heap. The r_get_put rule requires that we have a region in the current region context to read a boxed value. The r_letr rule introduces a new region into the region context. The r_only rule replaces the current context with a new one and continues evaluating the expression. The r_dealloc rule non-deterministically removes regions which are no-longer needed by the current program. Appendix B formally defines the notion of a free region variable.

A region is syntactically dead when it is not a free region variable of the program being evaluated. For well typed programs we can prove it is safe to always apply the dealloc_rule after evaluating the body of a r_letr expression, since our typing rules prevent a region variable from escaping the lexical scope of a `letr` expression.

The r_only rules replaces the current region context with a new one which must be a subset of the current region context because of our typing rules. Those regions that are not in the set Δ' can be safely deallocated. Even with r_dealloc rule the r_only rule is not redundant. Since our type system guarantees that body of the `only` never returns we can ignore any free region variables in the surrounding control context and just continue evaluating the body with the region variables, Δ', mentioned explicitly in the `only` expression.

2.2 Static Semantics of λ_ρ

Figure 2.2 describes the static semantics of λ_ρ. Again we assume that all bound region variables are uniquely named

$$
\begin{array}{rcl}
v & ::= & n \mid (\lambda x : s.e) \mid (\Lambda\rho.e) \mid (\mathtt{put}[\rho]\, v) \mid (\mathtt{exit}\, v) \\
E & ::= & [\,] \mid (+1\, E) \mid (-1\, E) \mid (\mathtt{if0}\, E\, e_1\, e_2) \mid \\
 & & (E\, e) \mid (v\, E) \mid (E\, [\rho]) \mid \\
 & & (\mathtt{put}[\rho]\, E) \mid (\mathtt{get}[\rho]\, E) \mid (\mathtt{exit}\, E) \\
P & ::= & \langle \Delta,\, e \rangle
\end{array}
$$

$$
\begin{array}{lrcl}
\text{r\_succ} & \langle \Delta,\, E[(+1\, n)] \rangle & \longmapsto & \langle \Delta,\, E[n+1] \rangle \\
\text{r\_pred} & \langle \Delta,\, E[(-1\, n)] \rangle & \longmapsto & \langle \Delta,\, E[n-1] \rangle \\
\text{r\_if\_then} & \langle \Delta,\, E[(\mathtt{if0}\, n\, e_1\, e_2)] \rangle & \longmapsto & \langle \Delta,\, E[e_1] \rangle \text{ where } n = 0 \\
\text{r\_if\_else} & \langle \Delta,\, E[(\mathtt{if0}\, n\, e_1\, e_2)] \rangle & \longmapsto & \langle \Delta,\, E[e_2] \rangle \text{ where } n \neq 0 \\
\text{r\_app} & \langle \Delta,\, E[((\lambda x : s.e)\, v)] \rangle & \longmapsto & \langle \Delta,\, E[e[x := v]] \rangle \\
\text{r\_rho\_app} & \langle \Delta,\, E[((\Lambda\rho.e)\, [\rho'])] \rangle & \longmapsto & \langle \Delta,\, E[e[\rho := \rho']] \rangle \\
\text{r\_fix} & \langle \Delta,\, E[(\mu\, f : s.e)] \rangle & \longmapsto & \langle \Delta,\, E[e[f := (\mu\, f : s.e)]] \rangle \\
\text{r\_exit} & \langle \Delta,\, E[(\mathtt{exit}\, v)] \rangle & \longmapsto & \langle \Delta,\, (\mathtt{exit}\, v) \rangle \\
\text{r\_get\_put} & \langle \Delta,\, E[(\mathtt{get}[\rho]\, (\mathtt{put}[\rho]\, v))] \rangle & \longmapsto & \langle \Delta,\, E[v] \rangle \text{ where } \rho \in \Delta \\
\text{r\_letr} & \langle \Delta,\, E[(\mathtt{letr}\, \rho\, \mathtt{in}\, e)] \rangle & \longmapsto & \langle \Delta \cup \{\rho\},\, E[e] \rangle \\
\text{r\_only} & \langle \Delta,\, E[(\mathtt{only}\, \Delta'\, e)] \rangle & \longmapsto & \langle \Delta',\, e \rangle \\
\text{r\_dealloc} & \langle \Delta,\, e \rangle & \longmapsto & \langle \Delta \setminus \{\rho\},\, e \rangle \text{ where } \rho \notin FRV(e)
\end{array}
$$

N.B. Assume that all bound region variables are uniquely named and that substitution preserves this property.

Figure 2: Operational Semantics

$$\frac{}{\Delta;\Gamma \vdash x : \Gamma(x)}\,\text{t\_var} \qquad \frac{}{\Delta;\Gamma \vdash n : \mathtt{int}}\,\text{t\_const}$$

$$\frac{\Delta;\Gamma \vdash e : \mathtt{int}}{\Delta;\Gamma \vdash (+1\, e) : \mathtt{int}}\,\text{t\_succ} \qquad \frac{\Delta;\Gamma \vdash e : \mathtt{int}}{\Delta;\Gamma \vdash (-1\, e) : \mathtt{int}}\,\text{t\_pred}$$

$$\frac{\Delta;\Gamma \vdash e : \mathtt{int} \qquad \Delta;\Gamma \vdash e_1 : \mathtt{s} \qquad \Delta;\Gamma \vdash e_2 : \mathtt{s}}{\Delta;\Gamma \vdash (\mathtt{if0}\, e\, e_1\, e_2) : s}\,\text{t\_if0}$$

$$\frac{\Delta';\Gamma, x : s \vdash e : t \qquad \Delta' \subseteq \Delta}{\Delta;\Gamma \vdash (\lambda x : s.e) : s \xrightarrow{\Delta'} t}\,\text{t\_lam}$$

$$\frac{\Delta;\Gamma \vdash e_1 : s \xrightarrow{\Delta'} t \qquad \Delta;\Gamma \vdash e_2 : t \qquad \Delta' \subseteq \Delta}{\Delta;\Gamma \vdash (e_1\, e_2) : s}\,\text{t\_app}$$

$$\frac{\Delta \cup \{\rho\};\Gamma \vdash e : s}{\Delta;\Gamma \vdash (\Lambda\rho.e) : \forall\rho.s}\,\text{t\_rho\_abs} \qquad \frac{\Delta;\Gamma \vdash e : \forall\rho.s \qquad \rho' \in \Delta}{\Delta;\Gamma \vdash (e\, [\rho']) : s[\rho := \rho']}\,\text{t\_rho\_app}$$

$$\frac{\Delta \cup \{\rho\};\Gamma \vdash e : s \qquad \rho \notin FRV(s)}{\Delta;\Gamma \vdash (\mathtt{letr}\, \rho\, \mathtt{in}\, e) : s}\,\text{t\_letr}$$

$$\frac{\Delta';\Gamma \vdash e : \mathtt{Ans} \qquad \Delta' \subseteq \Delta}{\Delta;\Gamma \vdash (\mathtt{only}\, \Delta'\, e) : \mathtt{Ans}}\,\text{t\_only}$$

$$\frac{\Delta;\Gamma \vdash e : s \qquad \rho \in \Delta}{\Delta;\Gamma \vdash (\mathtt{put}[\rho]\, e) : (s\ \mathtt{at}\ \rho)}\,\text{t\_put} \qquad \frac{\Delta;\Gamma \vdash e : (s\ \mathtt{at}\ \rho) \qquad \rho \in \Delta}{\Delta;\Gamma \vdash (\mathtt{get}[\rho]\, e) : s}\,\text{t\_get}$$

$$\frac{\Delta;\Gamma, f : s \vdash e : s}{\Delta;\Gamma \vdash (\mu f : s.e) : s}\,\text{t\_fix} \qquad \frac{\Delta;\Gamma \vdash e : s}{\Delta;\Gamma \vdash (\mathtt{exit}\, e) : \mathtt{Ans}}\,\text{t\_exit}$$

N.B. Assume that all bound region variables are uniquely named and that substitution preserves this property.

Figure 3: Static Semantics

and that substitution preserves this property. The judgment $\Delta; \Gamma \vdash e : s$ means under region context Δ and typing context Γ expression *e* has type *s*. These typing rules are similar to the "naive" rules of the Crary et al. calculus that ignore issues of region aliasing, where Δ act as a static capability context. Because of our implicit deallocation approach region aliasing does not result in unsoundness.

2.3 Examples

Figure 4 shows how a simple program is progressively refined to arrive at a fully region explicit version. The original program is a simple program that counts down from 10 and exits. From the original program we can derive a version that assumes integers are actually boxed values, and we can make the boxing and unboxing explicit. We can translate the program with explicit boxing and unboxing into an equivalent λ_ρ program. In λ_ρ the boxed integer type is represented as the type $(\texttt{int at } \rho)$. The `box` and `unbox` primitives are translated into `put` and `get` expressions respectively. The *cnt* function itself takes the region where to box the value as a *region parameter*. Finally, we account for the space of the closure needed to hold the actual function *cnt*, and place the closure in a separate region from the integer argument. Leaving us with a program similar to an example from Crary et al.[4].

Notice the type of the function *cnt* in the final version is annotated with the effect set $\{\rho, \rho_1\}$. It reads its argument from region ρ and reads region ρ_1 since it must access its own closure allocated in ρ_1. We refer to ρ as a *region parameter* since it is bound by a region abstraction and not a `letr`.

All though our operational model is not detailed enough to argue formally about space usage, we can informally reason about the space usage of the final program. On each iteration of *cnt* it puts a new integer in region ρ_2 until it terminates with the value $(\texttt{put}[\rho_2]\ 0)$. Notice that after each iteration the old argument is garbage, but we never reclaim any of the space used by the old arguments because they are allocated in region ρ_2, but on exit region ρ_2 is still live, since we return a boxed value as the result of the program. If we allocate each argument in a fresh region we can then free the old region which contains the old argument. Figure 5 demonstrates this optimization.

Free Early. Figure 5 (a) is a more space efficient version that copies the old argument into a new region, ρ', and then implicitly frees the old region where the old argument was allocated. Since ρ is a region parameter bound by a region abstraction, at runtime it may be aliased to another region variable, so we cannot statically determine whether it is safe to free the region associated with ρ. However, since it is not mentioned in the set of regions in the `only` expression it may be freed if at runtime it has not been aliased to ρ_1 or ρ'. We know statically it cannot be aliased to ρ', but it maybe aliased to ρ_1. In this case we will discover at runtime that it is safe to deallocate the region bound to ρ.

Region Aliasing. Figure 5 (b) demonstrates the case when we cannot free the region associated to the region parameter ρ, because on the first iteration of *cnt* region parameter ρ will be aliased to ρ_1 which is needed to store the closure of *cnt*. Because of our implicit deallocation approach our operational semantics will not get "stuck" even in the presence of this aliasing. On the subsequent iterations of *cnt* the region parameter ρ will be bound to a region which can be safely deallocated.

The optimization performed above is a kind of generalized tail-call optimization, which cannot be expressed in the standard Tofte-Talpin region calculus. Implementations of the Tofte-Talpin system are able to achieve something similar through storage mode analysis. However, storage mode analysis is unable to perform the specific optimization above, which is key to implementing a garbage collector with regions as we shall see. We can perform this optimization in more expressive region calculi, but in order to do so we must reason statically about aliasing to avoid unsoundness. In the presence of aliasing even these more expressive systems would simply forbid our final example, even though aliasing is only a problem for the first iteration.

3 Implementing a Realistic System

So far we have presented a novel region calculus that has several properties useful in the implementation of a garbage collector. Unfortunately, there are several other details that need to be addressed. Here we briefly address the remaining issues.

3.1 Invoking the Garbage Collector

The previous examples represent the basic approach to implementing a garbage collector on top of regions. Figure 6 fleshes out the idea in more detail. A compiler will transform a source level program into a region-annotated version in CPS form where at each potential garbage collection point there is a test of some counter to see if we should invoke the collector and copy the argument of the current continuation into a new region and free the old region, otherwise continue with the old region. Notice the use of region polymorphism for the continuation *k*, and that the typing of the conditional is dependent on *k* having a return type of `Ans`.

```
(let cnt =
 (μ cnt : int → Ans.
 (λn : int.
 (if0 n (exit n)
  (cnt (−1 n))))) in
(cnt 10))
```

(a) Original Program

```
(let cnt =
 (μ cnt : boxedint → Ans.
 (λn : boxedint.
 (let n′ = (unbox n) in
 (if0 n′ (exit n)
  (cnt (box (−1 n′)))))))) in
(cnt (box 10)))
```

(b) Explicit Boxing/Unboxing

```
(letr ρ′ in (let cnt =
 (μ cnt : (∀ρ.(int at ρ) →{ρ} Ans).
 (Λρ.(λn : (int at ρ).
 (let n′ = (get[ρ] n) in
  (if0 n′ (exit n)
   ((cnt [ρ]) (put[ρ] (−1 n′))))))))) in
((cnt [ρ′]) (put[ρ′] 10))))
```

(c) Translation into λ_ρ

```
(letr ρ1 in (letr ρ2 in (let cnt =
 (μ cnt : ((∀ρ.(int at ρ) →{ρ,ρ1} Ans) at ρ1).
 (put[ρ1] (Λρ.(λn : (int at ρ).
 (let n′ = (get[ρ] n) in
  (if0 n′ (exit n)
   (((get[ρ1] cnt) [ρ]) (put[ρ] (−1 n′))))))))) in
(((get[ρ1] cnt) [ρ2]) (put[ρ2] 10)))))
```

(d) Accounting for Closures

Figure 4: Simple Region Program

```
(letr ρ1 in (letr ρ2 in (let cnt =
 (μ cnt : ((∀ρ.(int at ρ) →{ρ,ρ1} Ans) at ρ1).
 (put[ρ1] (Λρ.(λn : (int at ρ).
 (let n′ = (get[ρ] n) in
  (if0 n′ (exit n)
   (letr ρ′ in (only {ρ′, ρ1}
    (((get[ρ1] cnt) [ρ′]) (put[ρ′] (−1 n′))))))))))) in
(((get[ρ1] cnt) [ρ2]) (put[ρ2] 10)))))
```

(a) Without Region Aliasing

```
(letr ρ1 in (let cnt =
 (μ cnt : ((∀ρ.(int at ρ) →{ρ,ρ1} Ans) at ρ1).
 (put[ρ1] (Λρ.(λn : (int at ρ).
 (let n′ = (get[ρ] n) in
  (if0 n′ (exit n)
   (letr ρ′ in (only {ρ′, ρ1}
    (((get[ρ1] cnt) [ρ′]) (put[ρ′] (−1 n′))))))))))) in
(((get[ρ1] cnt) [ρ1]) ))(put[ρ1] 10))
```

(b) With Region Aliasing

Figure 5: Space Efficient Region Program

```
...
(let copy : ∀ρ.∀ρ'(s at ρ) -{ρ,ρ'}-> (s at ρ') = ... in
(let k : ∀ρ.(s at ρ) -{ρ}-> Ans = ... in
(let x : (s at ρ) = ... in
(if0 limit
 (letr ρ' in
  (let x' : (s at ρ') = (((copy [ρ]) [ρ']) x) in
  (only {ρ'} ((k [ρ']) x'))))
 ((k [ρ]) x)))))
```

Figure 6: Invoking Garbage Collection

3.2 Intensional Polymorphism

In our example above we invoke a region-polymorphic copy function that takes a boxed value in region ρ and copies it into ρ'. Depending on the source language, the approach we have outlined so far may or may not be satisfactory. In particular we will need a copy function for each different type of continuation argument. Also the type of *copy* only guarantees a shallow copy of the value; if the type of the continuation argument, *s*, contains references to ρ, than our fragment above would not be well typed. Ideally, we would like one universal copy function with the following intensional type[8].

$$gc\_copy : \forall\alpha.\forall\rho.\forall\rho'.\alpha \xrightarrow{\{\rho,\rho'\}} (\alpha[\rho' := \rho])$$

Intensional type analysis allows for the definition of primitive recursive functions from types to types. In the type of *gc_copy* we use intensional type analysis to substitute one region variable for another. Intensional type analysis is usually expressed with a `Typerec` type constructor that encodes a primitive recursive function by structural induction on the structure of types. Here we simply use standard substitution notation, a trivial use of `Typerec`, for clarity. The type of *gc_copy* guarantees every value in region ρ is copied into region ρ'. Intensional type analysis can also be used to deal with closures.

3.3 Closures

So far we have ignored the issues of how to scan closures in a type-safe way. Many type-based compilers use existential types to abstract the type of the record containing free variables in a closure[9].

$$\exists\alpha.\langle\alpha, \langle\alpha, \texttt{int}\rangle \rightarrow \texttt{int}\rangle$$

Unfortunately, this abstraction prevents us from properly scanning and copying closures. Depending on the calculus, intensional type analysis may or may not be able to inspect the structure of the closure. In practice, type-based compilers do not actually enforce this abstraction, since exact garbage collectors need this information. Typically the compiler emits a pointer map describing the structure of the closure so that an exact collector can properly scan the closure. The pointer map is external to the underlying typed intermediate language, and typically there is no way to verify that it actually corresponds to the actual type of the closure. Tolmach presents a closure conversion technique that represents closures in an "interpreted style" due to Reynolds[18]. His approach avoids existentials and uses standard algebraic types with an explicit dispatch function to handle closures. See Figure 7.

Tolmach argues that this approach has several benefits, besides keeping the type system simple. In particular this transformation amounts to a simple type based closure analysis. Tolmach also outlines several other optimizations and a method that maintains separate compilation that make this approach seem extremely attractive. However, his separate compilation approach has some performance penalties.

The interpreted style of closure conversion amounts to encoding the information that is traditionally passed through pointer maps, directly in the type system. This makes a previous implicit interface between the compiler and garbage collector explicit. If we take the existential approach with a sufficiently powerful intensional analysis the runtime type information passed by the compiler will substitute for pointer maps. It's not clear which approach to closures will be more appropriate in practice.

All garbage collectors we have described have been "tagless" in that the compiler does not need to tag every value with extra type information. However, because we are using intensional type analysis the compiler must be passing runtime type information. The garbage collector is using the same information to implement its scanning functions. We can obtain a truly tagless system if we are able to monomorphize our code and pay a code blowup cost.

3.4 Pointer Sharing

In practice a realistic garbage collector needs to handle cyclic graphs and preserve pointer sharing to guarantee termination, deal with references, and avoid potentially exponential space blow ups.

Using a Hash Table. Sharing is preserved by using *forwarding pointers*, which turns a naive copy function into one that memoizes its arguments by mutating objects inplace. The in-place update overwrites live reachable data which complicates reasoning about soundness. A simpler approach whose soundness is obvious, is to avoid destructive update and use a hash table that hashes point-

```
let
  val y = 1
  val f =
      if e then (fn x:int => x)
           else (fn x:int => y)
in f 1
end
```

```
datatype clos =
   C1 of unit |  C2 of int

fun apply (C1 _) x = x
  | apply (C2 y) x = y

let
  val y = 1
  val f =
      if e then (C1 ())
           else (C2 y)
in apply f 1
end
```

Figure 7: Interpreted Style of Closure Conversion

$$
\begin{array}{rcl}
\mathit{empty} & : & \forall\rho.\langle\rangle \xrightarrow{\{\rho\}} (\rho\ \mathtt{at}\ (\rho_d,\ \rho_r)\ \mathtt{map}) \\
\mathit{insert} & : & \forall\alpha.\forall\rho.\forall\rho_d.\forall\rho_r.\langle(\rho\ \mathtt{at}\ (\rho_d,\ \rho_r)\ \mathtt{map}),\ (\alpha\ \mathtt{at}\ \rho_d),\ (\alpha\ \mathtt{at}\ \rho_r)\rangle \xrightarrow{\{\rho,\rho_d,\rho_r\}} \langle\rangle \\
\mathit{lookup} & : & \forall\alpha.\forall\rho.\forall\rho_d.\forall\rho_r.\langle(\rho\ \mathtt{at}\ (\rho_d,\ \rho_r)\ \mathtt{map}),\ (\alpha\ \mathtt{at}\ \rho_d)\rangle \xrightarrow{\{\rho,\rho_d,\rho_r\}} (\alpha\ \mathtt{at}\ \rho_r)\ \mathtt{opt}
\end{array}
$$

Figure 8: Trusted Hash Table Primitives

ers to pointers. This requires no modifications to our type system other than the addition of a new primitive $(\rho_d,\ \rho_r)\ \mathtt{map}$ which maps pointers to values in region ρ_d to pointers values in region ρ_r.

Figure 8 provides the interface to the hash table. The *empty* primitive takes no arguments and creates and empty hash table in region ρ, all the other primitives will allocate any needed space for this hash table in this region. The *insert* function adds a new entry in the hash table for boxed values of type α. Our *lookup* function searches the table for an entry with the same pointer value as its second argument returning the either pointer bound to it or its third argument or a special value if it is not found. Implementing a garbage collector that preservers sharing is simply a matter of programming.

Update with Extra Space. The hash table approach requires no serious modifications to our existing typing system and the soundness is quite obvious. However, it adds a significant amount of extra code to the trusted computing base. It also has some potential performance penalties in terms of time and space. We can get better performance in terms of time if we simply allow for null pointers and update of values. Figure 9 provides a signature for the new primitives.

A boxed value of type α in region ρ that may be null has the type $((\alpha\ \mathtt{at}\ \rho)\ \mathtt{opt})$. The constant *null* is the null pointer and the primitive *value* injects a non-null pointer into the opt type. The *getVal* primitive tests for a null pointer and returns the value if it is not null or its second argument if it is null. The update primitive destructively updates a pointer with a new value.

$$
\begin{array}{rcl}
\mathit{nil} & : & \forall\alpha.\forall\rho.(\alpha\ \mathtt{at}\ \rho)\ \mathtt{opt} \\
\mathit{value} & : & \forall\alpha.\forall\rho.(\alpha\ \mathtt{at}\ \rho) \rightarrow (\alpha\ \mathtt{at}\ \rho)\ \mathtt{opt} \\
\mathit{getVal} & : & \forall\alpha.\forall\rho.\langle(\alpha\ \mathtt{at}\ \rho)\ \mathtt{opt},\ (\alpha\ \mathtt{at}\ \rho)\rangle \rightarrow (\alpha\ \mathtt{at}\ \rho) \\
\\
\mathit{update} & : & \forall\alpha.\forall\rho.\langle(\alpha\ \mathtt{at}\ \rho),\ \alpha\rangle \xrightarrow{\{\rho\}} \langle\rangle \\
\\
 & & (\alpha,\ \rho)\ \mathtt{obj} \equiv \langle(\alpha\ \mathtt{at}\ \rho)\ \mathtt{opt},\ \alpha\rangle
\end{array}
$$

Figure 9: Forwarding with Options and Update

Given these primitives we can define an $(\rho,\ \alpha)\ \mathtt{obj}$ which is actually an abbreviation for the pair $\langle(\alpha\ \mathtt{at}\ \rho)\ \mathtt{opt},\ \alpha\rangle$. An unboxed integer which maybe forwarded to region ρ would have type $(\mathtt{int},\ \rho)\ \mathtt{obj}$. A boxed integer object in region ρ' which may be forwarded to region ρ would have type $((\mathtt{int}\ \mathtt{at}\ \rho'),\ \rho)\ \mathtt{obj}$. We also need to change the type of our *gc_copy* function to

have the following intensional type

$$gc\_copy : \forall\alpha.\forall\rho.\forall\rho'.\forall\rho''.\alpha \xrightarrow{\{\rho,\rho',\rho''\}} (\alpha[\rho,\rho' := \rho',\rho''])$$

where $\alpha[\rho'',\rho' := \rho',\rho]$ denotes the simultaneous substitution of ρ and ρ' with ρ' and ρ'' respectfully.

The mutator code only looks at the second component of this pair while the first component can be used by the garbage collector to store forwarding pointers. Whenever an object is created the mutator and collector should initialize the first field to null. This correctness requirement is not captured in the type system. Unfortunately this approach requires that mutator code be run in the lexical scope of the region that will contain forwarded objects in the future, so both the current allocation space and the future allocation space must be statically live at the same time, but since the future allocation space will contain no objects, there need not be any significant space penalty.

4 Conclusions

Efficiency From an algorithmic standpoint the approaches outlined above are competitive with traditional unsafe collectors. However, we currently are not able to safely encode collection algorithms that replace pointers to values in one region with pointers to values in another. Our approach does not require any more type information than tag-free approaches[17]. The only extra cost is the scanning required for implicit deallocation of regions.

Interface to Garbage Collectors. Most garbage collectors are very closely tied to a particular code generator that maintains data-representation invariants that the garbage collector needs to operate safely. Changes in data-representations used by the code generator may require changes to garbage collectors and vice versa. Mismatches between the code-generator and garbage collector are the source of many hard-to-track down bugs in practice.

Also notice that the traditional distinction between the *mutator* and *collector* disappears. The collector is just a term that is indistinguishable from the any other term in the language. Optimizing compilers can perform optimizations on a whole program that includes the collector. Along with reducing the *trusted computing base* (TCB) our approach allows one to formalize many data-representation invariants explicitly in a well defined type system, such that any type-preserving transformation maintains garbage collector safety.

Being able to formalize these invariants has a very big software engineering advantage since more bugs can be caught early on in the development cycle. These invariants can be used to aid the generation of Proof Carrying Code (PCC) and Typed Assembly Language (TAL) [12, 11], opening the potential of system that can verify the safety of an entire program, without assuming the existence of a trusted garbage collector. Currently these systems use conservative collectors to reclaim storage.

Unlike, other region calculi, ours does not make any a priori assumptions about which values are boxed and unboxed. Values of type `int` are unboxed integers while values of type $(\texttt{int at } \rho)$ are integers boxed in region ρ. This explicit distinction also simplifies our proofs, since the core calculus closely resembles standard typed lambda calculi. This approach also demonstrates that is is easy to extend existing typed lambda calculi used in compilers with a notion of regions.

Implicit Deallocation. One reason Aiken et al. [1] and Crary et al. [4] require a much more complicated static analysis is that they are trying to guarantee statically that it is safe to explicitly deallocate a region. This is undecidable because of *region aliasing*, which causes regions that are statically distinct to become, at run time, aliased to the same value. Both systems try to address the region aliasing problem with conservative static approaches. Crary et al. track "uniqueness" information. Aiken et al. solve a system of constraints over program control flow. These approaches allow for deallocation to be a cheap constant time operation. Our system is immune to the problems associated with region aliasing, because our implicit approach will only deallocate regions which are known to be syntactically dead at run time.

Related Work. The system described by Aiken et al.[1] is expressive enough to implement our idea but requires global analysis to guarantee safety. The system of Crary [4] is also sufficiently expressive, and does not require a global analysis to verify safety. However, their type system is much more complicated. If we were to use the Crary system we would be left with a small runtime TCB but a much more complicated static checker to verify the static safety properties.

To improve memory utilization of a region-based system, one can integrate a trace-based garbage collector with region managed memory [7]. Moreover an idiom seen in some region schemes refered to as *double copying* [15] is basically an explicit two-space copying garbage collector implemented on top of a safe region-based scheme. Adding a collector to a region system improves memory utilization, but it does not allow us to maintain a small TCB.

There exists a safe runtime variant of regions that uses dynamic reference counting [5] which has all the benefits of static regions as well as good separate compilation properties. However it requires a change in programming

model and leaves the burden of deallocation to the programmer. Explicit regions provide a simple safe and efficient manual allocation mechanism to the programmer.

Future Work. Because our types track the location of an object, updating a pointer to a value in one region with a pointer to a value in another is an unsound operation. We are investigating approaches that will allow this so we can remove the overhead of having to reserve space for forwarding pointers, since many unsafe garbage collectors perform this optimization. Being able to update pointers in this way is also important for implementing *generational collection* schemes. Ideally a sound type system that can handle the above will allow us to encode many of the pointer/non-pointer store invariants needed to implement write barriers for generational systems. We intend to implement a prototype system using the techniques outlined so far to better understand the performance properties of the various approaches.

References

[1] AIKEN, A., FÄHNDRICH, M., AND LEVIEN, R. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)* (La Jolla, California, 18–21 June 1995), pp. 174–185. *SIGPLAN Notices* 30(6), June 1995.

[2] BANERJEE, A., HEINTZE, N., AND RIECKE, J. G. Region analysis and the polymorphic lambda calculus. In *Proceedings, Fourteenth Annual IEEE Symposium on Logic in Computer Science* (Trento, Italy, 2–5 July 1999), IEEE Computer Society Press, pp. 88–97.

[3] BIRKEDAL, L., TOFTE, M., AND VEJLSTRUP, M. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1996), ACM Press, pp. 171–183.

[4] CRARY, K., WALKER, D., AND MORRISETT, G. Typed memory management in a calculus of capabilities. In *Principles of Programming Languages* (San Antonio, TX, Jan. 1999), pp. 262–275.

[5] GAY, D., AND AIKEN, A. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)* (Montreal, Canada, 17–19 June 1998), pp. 313–323. *SIGPLAN Notices* 33(5), May 1998.

[6] GUTTMAN, J., RAMSDELL, J., AND WAND, M. Vlisp: A verified implementation of scheme. *Lisp and Symbolic Computation* (94).

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

[8] HARPER, R., AND MORRISETT, G. Compiling polymorphism using intensional type analysis. In *Principles of Programming Languages* (San Francisco, Jan. 1995).

[9] MINAMIDE, Y., MORRISETT, G., AND HARPER, R. Typed closure conversion. In *Principles of Programming Languages* (1996), pp. 271–283.

[10] MORRISETT, G., FELLEISEN, M., AND HARPER, R. Abstract models of memory management. In *Functional Programming and Computer Architecture* (San Diego, 1995).

[11] MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. From System F to typed assembly language. In *POPL '98: 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1998), ACM Press, pp. 85–97.

[12] NECULA, G. Proof-carrying code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, Jan. 1997), ACM Press, pp. 106–119.

[13] SHAO, Z. Flexible representation analysis. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming* (Amsterdam, The Netherlands, 9–11 June 1997), pp. 85–98.

[14] TARDITI, D., MORRISETT, G., CHENG, P., STONE, C., HARPER, R., AND LEE, P. TIL: A type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, 21–24 May 1996), pp. 181–192. *SIGPLAN Notices* 31(5), May 1996.

[15] TOFTE, M., BIRKEDAL, L., ELSMAN, M., HALLENBERG, N., OLESEN, T. H., SESTOFT, P., AND BERTELSEN, P. Programming with reigons in the ML Kit (for version 3). Tech. Rep. DIKU-TR-98/25, University of Copenhagen, December 1998.

[16] TOFTE, M., AND TALPIN, J.-P. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings from the 21st annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1994).

[17] TOLMACH, A. Tag-free garbage collection using explicit type parameters. *LISP Pointers 7*, 3 (July-Sept. 1994), 1–11.

[18] TOLMACH, A., AND OLIVA, D. P. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming 8*, 4 (July 1998), 367–412.

[19] WRIGHT, AND FELLEISEN. A syntactic approach to type soundness. *Information and Computation (formerly Information and Control) 115* (1994).

A Type Soundness of λ_ρ

A well typed program is in a *stuck* state if from $\langle \Delta, e\rangle$ there does not exist $\langle \Delta', e'\rangle$ such that $\langle \Delta, e\rangle \longmapsto \langle \Delta', e'\rangle$ and e is not a value.

Theorem 1 (Type Soundness) *If* $\Delta;\Gamma \vdash e : s$ *and* $\langle \Delta, e\rangle \longmapsto^* \langle \Delta', e'\rangle$ *then* $\langle \Delta', e'\rangle$ *is not a stuck state.*

Proof: (Sketch) By Subject Reduction and Progress Lemmas.

Lemma 1.1 (Subject Reduction) *If* $\Delta;\Gamma \vdash e : s$ *and* $\langle \Delta, e\rangle \longmapsto \langle \Delta', e'\rangle$ *then* $\Delta';\Gamma \vdash e' : s$

Proof: (Sketch)

Case r_get_put $\langle \Delta, E[(\texttt{get}[\rho]\ (\texttt{put}[\rho]\ v))]\rangle \longmapsto \langle \Delta, E[v]\rangle$ where $\rho \in \Delta$
By assumption $\Delta;\Gamma \vdash E[(\texttt{get}[\rho]\ (\texttt{put}[\rho]\ v))] : t$. By induction on the typing derivation and inspection of the t_put rule $\Delta;\Gamma \vdash E[v] : t$.

Case r_letr $\langle \Delta, E[(\texttt{letr}\ \rho\ \texttt{in}\ e)]\rangle \longmapsto \langle \Delta \cup \{\rho\}, E[e]\rangle$
By assumption $\Delta;\Gamma \vdash E[(\texttt{letr}\ \rho\ \texttt{in}\ e)] : t$. By induction on the typing derivation and inspection of the t_letr typing judgement we conclude that $\Delta \cup \{\rho\};\Gamma \vdash E[e] : t$.

Case r_only $\langle \Delta, E[(\texttt{only}\ \Delta'\ e)]\rangle \longmapsto \langle \Delta', e\rangle$
By assumption $\Delta;\Gamma \vdash E[(\texttt{only}\ \Delta'\ e)] : \texttt{Ans}$. By induction of the typign derivation and inspection of the t_only rule $\Delta';\Gamma \vdash e : \texttt{Ans}$.

Case r_dealloc $\langle \Delta, e\rangle \longmapsto \langle \Delta \setminus \{\rho\}, e\rangle$ where $\rho \notin FRV(e)$
By assumption $\Delta;\Gamma \vdash e : t$ and since $\rho \notin FRV(e)$. Since $\rho \notin FRV(e)$ then $\Delta \setminus \{\rho\};\Gamma \vdash e : t$ by context strengthening.

All other cases are standard.

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

- *There exists* $\langle \Delta', e'\rangle$ *such that* $\langle \Delta, e\rangle \longmapsto \langle \Delta', e'\rangle$, *or*
- *e is a value.*

Proof: (Sketch) By structural induction on expressions.

B Free Region Variables

$$
\begin{array}{rcl}
FRV(\texttt{int}) & = & \{\} \\
FRV(s \xrightarrow{\Delta} t) & = & FRV(s) \cup FRV(t) \cup \Delta \\
FRV(\forall \rho.s) & = & FRV(s) \setminus \{\rho\} \\
FRV((s\ \texttt{at}\ \rho)) & = & FRV(s) \cup \{\rho\} \\
 & & \\
FRV(\texttt{Ans}) & = & \{\} \\
FRV(x) & = & \{\} \\
FRV(n) & = & \{\} \\
FRV((+1\ e)) & = & FRV(e) \\
FRV((-1\ e)) & = & FRV(e) \\
FRV((\texttt{if0}\ e\ e_1\ e_2)) & = & FRV(e) \cup FRV(e_1) \cup FRV(e_2) \\
FRV((\lambda x : s.e)) & = & FRV(e) \cup FRV(s) \\
FRV((e_1\ e_2)) & = & FRV(e_1) \cup FRV(e_2) \\
FRV((\Lambda \rho.e)) & = & FRV(e) \setminus \{\rho\} \\
FRV((e\ [\rho])) & = & FRV(e) \cup \{\rho\} \\
FRV((\texttt{letr}\ \rho\ \texttt{in}\ e)) & = & FRV(e) \setminus \{\rho\} \\
FRV((\texttt{only}\ \Delta\ e)) & = & FRV(e) \cup \Delta \\
FRV((\texttt{put}[\rho]\ e)) & = & FRV(e) \cup \{\rho\} \\
FRV((\texttt{get}[\rho]\ e)) & = & FRV(e) \cup \{\rho\} \\
FRV((\mu\ f : s.e)) & = & FRV(e) \cup FRV(s) \\
FRV((\texttt{exit}\ e)) & = & FRV(e)
\end{array}
$$