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 preserves the type safety of the program being garbage collected. We describe the technique in detail and report performance measurements for a few microbenchmarks. We also include a formal semantics for a novel region calculus that supports early deallocation, and prove the type soundness of the 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. 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 [28] 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 [27]. 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 [23, 15]. 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.

\(^1\)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

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” [13] 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 [16, 12] and other forms of ad-hoc polymorphism that use dictionary passing [31] 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. [18, 22, 29] 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 [29] 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 [12]. 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 memory management [28, 1, 11, 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. Our 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 formally describe a core language that 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 Region Calculus that Support Early Deallocation

Since the original Tofte-Talpin calculus there have been many different formulations of various region calculi [1, 11, 7]. For our purposes the region system need not be particularly advanced. We do not need to separate read and write effects, support effect polymorphism, or allow for dangling pointers. All of these features are included in the original Tofte-Talpin region calculus [28]. With the exception of [11] most of the various region calculi are formulated as type-and-effect systems. In these calculi every expression has associated with it an inferred set of effects which represents an estimate of the set of regions an expression may access. Rather than infer a set of effects [11] presents a region calculus where an expression is typed with respect to a “capability context” which explicitly restricts the set of regions a given expression may access. The primary difference between these two approaches is that the former is primarily concerned with finding efficient and precise effect annotations which lead to more memory efficient programs that do not require a garbage collector. Capability systems are designed to verify that effects inferred by such systems are correct. We take the capability view of regions since we are primarily interested in validating that a particular effect assignment is valid.

The original Tofte-Talpin calculus required that regions be allocated and deallocated in a strict LIFO fashion. That is the most recently allocated region must be the first region that is deallocated. The Aiken, Fähndrich, and Levien [1] present a static analyses that allows for early deallocation which allows for more flexible non-LIFO region allocation policies. The capability calculus of [11] also supports early deallocation. For our type-preserving garbage collector early deallocation is critically important. However, rather than simply using the capability calculus of [11] we derive a simpler and in some respects more expressive calculus that relies on dynamic checking to handle certain aliasing issues that the capability calculus deals with in a purely static way. We will discuss the aliasing issue later when we discuss our static semantics.

2Early in the sense of sooner than what a strict LIFO policy would allow.
types \( \tau \) ::= unit
\[ \tau_1 \xrightarrow{\Delta} \tau_2 \] function type with effect
\( \forall \rho. \tau \) region polymorphic type
(\( \tau \) at \( \rho \)) region annotated type

Ans

terms \( e \) ::= \( x \)
\( \langle \rangle \) effect annotated function
(\( e_1 e_2 \)) region abstraction
(\( e[\rho] \)) region application
let \( \rho \) in \( e \) allocate new region
put[\( \rho \)](\( e \)) put value in region
get[\( \rho \)](\( e \)) get value from region
only \( \Delta \) in \( e \) deallocate regions early
(\( \text{fix } f : \tau. v \))

values \( v \) ::= \( \langle \rangle \) | (\( \lambda x : \tau. e \))\( \Delta \) | (\( \Lambda \rho. e \)) | put[\( \rho \)](\( v \))

region contexts \( \Delta \) ::= \{ \} | \{ \rho_1, \ldots, \rho_n \}

Figure 3: Abstract Syntax

2.1 Syntax

Figure 3 describes the abstract syntax of a capability style region calculus. Previous region calculi explicitly annotate every type and term with a region annotation to precisely reflect a particular set of implementation details, such as the fact that most objects do not fit in machine registers and must have auxiliary storage allocated for them. Rather than “baking in” such a decision and cluttering our syntax we introduce two term-level operators and a new type constructor. One term-level operator is put[\( \rho \)](\( e \)), which evaluates its argument \( e \) and stores the resulting value into region \( \rho \). If the type of the argument \( e \) is of type \( \tau \) the type of the resulting value is (\( \tau \) at \( \rho \)). The operator get[\( \rho \)](\( e' \)) expects that its argument \( e' \) evaluates to a value of type (\( \tau \) at \( \rho \)). It then fetches the value from region \( \rho \) and returns a value of type \( \tau \). The terms put[\( \rho \)](\( e \)) and get[\( \rho \)](\( e \)) are inspired by the translation of [7], which translates a simplified effect based region calculus into a novel typed lambda calculus. In the translations the effects of the original region calculi are encoded as term level objects in the target lambda calculus. A value allocated in region \( \rho \) is represented by the syntactic term put[\( \rho \)](\( v \)). We treat region contexts \( \Delta \) as sets of region variables.

When the body of function (\( \lambda x : \tau. e \))\( \Delta \) captures values in its closure we must account for any regions needed to access those values by annotating the function with the set of region variables needed to access those values. Function types are annotated with a corresponding
set of region variables needed to evaluated the body of the function. Since we have an explicit region abstraction term \((\Lambda \rho.e)\) and a separate fix point term in our language we support polymorphic recursion over region variables. The term only \(\Delta\) in \(e\) is our construct to support early deallocation. Our type-preserving garbage collector will contain a mix of “direct style” and CPS converted expressions, for that reason we include a the type \(\text{Ans}\) which is the return type for continuations, so that we can distinguish between continuations and normal functions.

2.2 Dynamic Semantics

Figure 4 describes the dynamic semantics of our calculus in the style of Felleisen and Wright [34]. We assume that all bound region variables are unique and that substitution alpha renames variables to preserve this property. We introduce the notion of a region stack, ranged over by \(R\), which are a sequence of nested \(\text{letr}\) expressions ending in a hole ([ ]). The notation \(R[e]\) represents a region stack with the hole of the stack replaced by \(e\). \(E\) ranges over control contexts. The notation \(E[e]\) represents a control context with the hole of the control context replaced by \(e\). Program consists of a series of \(\text{letr}\) bindings establishing an initial region stack surrounding an expression to evaluate. Answers are a subset of program expressions, which consists either of a single \(\text{halt}\) expression or a region stack enclosing a value.

We define two one step reduction relations. The relation \(\mapsto_{e}\) performs local reductions on expressions, while the relation \(\mapsto_{P}\) performs global reductions on whole programs. The \(\mapsto_{e}\) relation is the standard one step reduction relation for a pure call-by-value lambda calculus. This local reduction relation is used in the definition of the global program reductions by the rule \(\text{rdspure}\). The global program reduction rule \(\text{rdsletr}\) hoists an inner \(\text{letr}\) binding to the top-level region stack. Since we assume all bound variables are unique, the region in the \(\text{letr}\) binding is guaranteed to be a fresh variable. The \(\text{rdget}\) rule converts a value allocated in region \(\rho\) to a pure value, if the region \(\rho\) is currently bound by the enclosing region stack. The \(\text{rdonly}\) rule throws away the surrounding control context and continues evaluating its body in a new region stack defined by the \(\text{only}\) expression. The notation \(R^\Delta\) is the smallest region stack binding the variables in \(\Delta\).

The \(\text{rdhalt}\) rule immediately throws away any non-trivial surrounding control context and region stack and reduces to an answer. The \(\text{rdfsfree}\) rule is a non-deterministic rule that removes unneeded region bindings from the region stack. A region binding is unneeded if removing the binding does not prevent rest of the program from remaining well typed. The relation \(\mapsto_{P}\) is the reflexive transitive closure of our single step program reduction relation. Notice \(\text{rdfsfree}\) implicitly allows for non-LIFO allocation policy, already providing support for early deallocation, which make make the \(\text{rdsonly}\) rule seem redundant. However, if we did not include the \(\text{rdsonly}\) rule we could not immediately reclaim regions that are bound in the useless surrounding control context of an expression for which we know will not return.
programs  $P ::= R[e]$  
answers  $A ::= R[v] \mid \text{halt}^\tau$  
control contexts  $E ::= \emptyset$  
\[ \mid (E \ e) \mid (v \ E) \mid (E[\rho]) \mid \text{put}[\rho](E) \mid \text{get}[\rho](E) \]  
region stacks  $R ::= \emptyset \mid \text{letr} \ \rho \ \text{in} \ R$  

Expression Reductions

\[ \text{rdsbetav} \ ((\lambda x: \tau.e)\Delta v) \mapsto e[v/x] \]
\[ \text{rdstapp} \ ((\Lambda \rho.e)[\rho']) \mapsto e[\rho'/\rho] \]
\[ \text{rdsfix} \ (\text{fix} : \tau.v) \mapsto v[(\text{fix} : \tau.v)/f] \]

Program Reductions

\[ \text{rdspure} \ R[E[e_1]] \mapsto_P R[E[e_2]] \text{ where } e_1 \mapsto e e_2 \]
\[ \text{rdsletr} \ R[E[\text{letr} \ \rho \ \text{in} \ e]] \mapsto_P R[\text{letr} \ \rho \ \text{in} \ E[e]] \]
\[ \text{rdsget} \ R[\text{letr} \ \rho \ \text{in} \ R'[E[\text{get}[\rho](\text{put}[\rho](v))]]] \mapsto_P R[\text{letr} \ \rho \ \text{in} \ R'[E[v]]] \]
\[ \text{rdsonly} \ R[E[\text{only} \ \Delta \ \text{in} \ e]] \mapsto_P R'[E[e]] \]
\[ \text{rdshalt} \ R[\text{letr} \ \rho \ \text{in} \ R'[e]] \mapsto_P \text{halt}^\tau \text{ where } E \neq [\ ] \]
\[ \text{rdsfreer} \ R[\text{letr} \ \rho \ \text{in} \ R'[e]] \mapsto_P R[\text{letr} \ \rho \ \text{in} \ R'[e]] \text{ where } \vdash R[R'[e]] \text{ wt} \]

\[ \emptyset \Delta \overset{\text{def}}{=} [\ ] \]
\[ (\text{letr} \ \rho \ \text{in} \ R)\Delta^{\omega(\rho)} \overset{\text{def}}{=} (\text{letr} \ \rho \ \text{in} \ R\Delta^{\omega(\rho)}) \]
\[ (\text{letr} \ \rho \ \text{in} \ R)^\Delta \overset{\text{def}}{=} R^\Delta \text{ where } \rho \notin \Delta \]

Multi-step Reduction

\[
\begin{align*}
\text{idref} & \quad \text{idref} \quad \text{idref} \\
\dfrac{P \mapsto_P^* P}{\text{mstprefl}} & \quad \dfrac{P_1 \mapsto_P^* P_2 \quad P_2 \mapsto_P^* P_3}{\text{mstptrans}} & \quad \dfrac{P_1 \mapsto_P^* P_2}{\text{mstprds}}
\end{align*}
\]

Figure 4: Dynamic Semantics
value environment  \( \Gamma := \{ \} \mid \{ x_1 : \tau_1, \ldots, x_n : \tau_n \} \)

<table>
<thead>
<tr>
<th>Judgement</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \vdash P ) ( \text{wt} )</td>
<td>( P ) is a well typed program.</td>
</tr>
<tr>
<td>( \Delta; \Gamma \vdash e : \tau )</td>
<td>( e ) has type ( \tau ) under ( \Delta; \Gamma ). We always implicitly require ( \Delta \vdash \Gamma ) ( \text{wfenv} )</td>
</tr>
<tr>
<td>( \Delta \vdash \tau ) ( \text{wf} )</td>
<td>( \tau ) is a well formed type with respect to ( \Delta ).</td>
</tr>
<tr>
<td>( \Delta \vdash \Gamma ) ( \text{wfenv} )</td>
<td>( \Gamma ) is a well formed value environment with respect to ( \Delta ).</td>
</tr>
</tbody>
</table>

Figure 5: Summary of Typing Judgements

2.3 Static Semantics

Figure 5 summarizes the main typing judgments for our static semantics. Figure 6 contains the inference rules for the main typing judgments while Figure 7 defines some auxiliary well formedness conditions. We use the notation of \( \Delta \uplus \Delta' \) to represent the union of two disjoint set of region variables, and the notation \( \Gamma \uplus \{ x : \tau \} \) to be the extension of an environment mapping the variable \( x \) to the type \( \tau \) where \( x \) does not occur already bound in \( \Gamma \).

The judgment \( \vdash P \) \( \text{wt} \) simply asserts that the closed program \( P \) has some valid type. It holds only if the program \( P \) is well typed under an empty typing environment. The judgment \( \Delta; \Gamma \vdash e : \tau \) asserts that the expression \( e \) has type \( \tau \) under the value environment \( \Gamma \) and the region context \( \Delta \). The judgment \( \Delta \vdash \tau \) \( \text{wf} \) asserts that all the free region variables in \( \tau \) occur in \( \Delta \). The judgment \( \Delta \vdash \Gamma \) \( \text{wfenv} \) generalizes this notion for value environments.

In Figure 6, to simplify our presentation of the rules for the judgment \( \Delta; \Gamma \vdash e : \tau \), we implicitly assume that region variables free in \( \Gamma \) occur in the region context \( \Delta \). That is we omit the explicit constraint that \( \Delta \vdash \Gamma \) \( \text{wfenv} \) for many of the rules. For the non-trivial rules such \( \text{htonly} \) and \( \text{htabs} \) we make the \( \Delta \vdash \Gamma \) \( \text{wfenv} \) constraints explicit for clarity. The explicit constraints also guarantee that our typing rules are deterministic by splitting the value environments and region contexts in a unique way.

The typing rules in Figure 6 closely resemble the typing rules for a polymorphic simply-typed lambda calculi, where type polymorphism has been replaced with region polymorphism. The rules \( \text{htabs} \) differs for the standard rule in that we check the body of the function in a value environment that contains a subset of the enclosing value environment. We check the body of a value environment consisting of those bindings whose free region variables are mentioned in the effect annotation for the function. A similar restriction is placed on the \( \text{htonly} \) rule. A simple inductive argument will establish the fact that any closed program of type \( \text{Ans} \) which evaluates to an answer must evaluate to the answer \( \text{halt}^{\text{Ans}} \). So informally speaking the requirement that the body of the \( \text{only} \) \( \Delta \) \( \text{in} \) \( e \) expression be of type \( \text{Ans} \) means the body does not return. Our dynamic semantics takes advantage of this fact by throwing away the unneeded control context and region stack.

Because the typing rules \( \text{htonly} \) and \( \text{htabs} \) check subexpressions in a non-standard way the standard substitution lemma for terms does not hold. A more restrictive substitution for values and \( \text{fix} \) expressions does hold. Since our calculus is call-by-value the more restrictive
Δ; Γ ⊨ e : τ

Figure 6: Main Typing Judgement
Figure 7: Auxiliary Typing Judgements
lemma is sufficient to prove type soundness for our calculus.

2.4 Early Deallocation

A realistic implementation of our abstract semantics must adopt a “region passing” semantics, where region variables are bound to dynamic values at runtime, and therefore region abstractions cannot be erased. The only expression takes an arbitrary set of region variables. At runtime we simply note what regions are dynamically bound to the only expression and safely deallocate any other regions, since they are not needed to evaluate the rest of the program. 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. The cost of the deallocation operation is at worst linearly related to the number of live regions. Region deallocation primitives in other systems are constant time operations. So our more flexible dynamic approach is not without its cost. Though for our type-preserving garbage collector this extra cost is negligible, since we can bound the number of live regions to a small constant.

Consider the function

```haskell
fun f[ρ_a, ρ_b](x:int at ρ_b):Ans =
  free_region ρ_a in (get[ρ_b](x) ; halt())
```

which uses a new free_region operator, which deallocates ρ_a before evaluating its body. At first glance it would seem that region ρ_a is not used in the body of f so this early deallocation of ρ_a is safe. However, consider the following calling context for f

```haskell
letr ρ_1, ρ_2 in
  if e then f[ρ_1, ρ_2](put[ρ_1](1))
  else f[ρ_1, ρ_1](put[ρ_1](1))
```

The expression put[ρ](1) stores the integer into the region ρ_1 and returns a reference to the integer. Notice that if we execute 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 deallocating region variables which maybe aliased through various typing disciplines [11]. 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

```haskell
fun f[ρ_a, ρ_b](x:int at ρ_b):Ans =
  only ρ_b in (get[ρ_b](x) ; halt())
```

At runtime we determine what region is actually bound to ρ_b, and deallocate the region bound to ρ_a if it is distinct from ρ_b. 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 simply reserve
a “live-bit” for each region and mark all regions bound in the context of the only expression
as live. All other unmarked regions can be reclaimed. This particular approach takes time
proportional to the number of total regions allocated regions in order to reclaim all the live
regions. An alternative implementation that uses doubly-linked lists, of free and allocated
regions can be implemented whose runtime is proportional to the number of live regions.

This dynamic approach to region deallocation is similar to the work of Aiken and Gay [14].
However, they use a relatively weak region type system and a more expensive reference count-
ing 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.

2.5 Safety Properties

The safety properties of our language are standard. We include the full proofs of all the
theorems and lemmas in Appendix A. Here we only describe the main theorem and associated
lemmas. We begin with the definition of stuck programs

**Definition 1 (Stuck Program)** The evaluation of a program \( P_1 \) is stuck if \( P_1 \) is not an
answer and there is no \( P_2 \) such that \( P_1 \xrightarrow{p} P_2 \).

The main safety theorem is as follows

**Theorem 1 (Type Soundness)** If \( \vdash P_1 \text{ wt} \) then, there is no stuck \( P_2 \) such that \( P_1 \xrightarrow{*} P_2 \).

The proof of our main theorem follows directly with a slightly stronger induction hypothesis
over the \( \xrightarrow{p} \) relation and the application of Lemma 1.1 and Lemma 1.2.

**Lemma 1.1 (Type Preservation of Programs)** If \( \vdash P_1 \text{ wt} \) and \( P_1 \xrightarrow{p} P_2 \), then \( \vdash P_2 \text{ wt} \).

**Lemma 1.2 (Progress)** If \( \vdash P_1 \text{ wt} \) then, there exists \( P_2 \) such that \( P_1 \xrightarrow{p} P_2 \) or \( P_2 \) is an
answer. i.e. \( P_1 \) is not stuck.

We do not have a general substitution lemma for terms but a restricted form which only
holds for values and fix expressions

**Lemma 1.3 (Typing Under Term Substitution)** If \( \Delta;\Gamma \vdash e:\tau \) and \( \Delta;\Gamma \cup \{ x:\tau \} \vdash e' : \tau' \)
then \( \Delta;\Gamma \vdash e'[e/x] : \tau' \), where \( e = v \) or \( e = (\text{fix } f : \tau''.v) \).

The following lemma allows us to throw away unneeded bindings and region variables from
our value environment and region context by inspecting the type of values and fix expressions
it is need in our proof of the substitution lemma for the \htabs case.

**Lemma 1.4 (Region Context Strengthening)** If \( \Delta' \vdash \tau \text{ wf} \), \( \Delta \vdash \Gamma \text{ wfenv} \), \( \Delta' \vdash \Gamma' \text{ wfenv} \), and \( \Delta \cup \Delta';\Gamma \cup \Gamma' \vdash e:\tau \) where \( e = v \) or \( e = (\text{fix } f : \tau''.v) \) then \( \Delta';\Gamma' \vdash e : \tau \)

The remaining lemmas are used in the proof of the previous lemmas and are included for
completeness, but are not technically interesting
Lemma 1.5 (Type Preservation of Expression) If \( \Delta; \emptyset \vdash e_1 : \tau \) and \( e_1 \mapsto e_2 \), then \( \Delta; \emptyset \vdash e_2 : \tau \).

Lemma 1.6 (Redux Decomposition) If \( \Delta; \emptyset \vdash e : \tau \) then \( e \) is a value or \( e = E[r] \) where \( r \) is a redux. A redux is any of the following forms:

1. \(((\lambda x: \tau'.e')^{\Delta''} v)\) where \( \Delta = \Delta' \uplus \Delta'' \)
2. \(((\Lambda e')[\tau'])\)
3. \((\text{fix} f: \tau'.v)\)
4. \(\text{let} r \in e'\)
5. \(\text{get}[\rho](\text{put}[\rho](e'))\)
6. only \( \Delta' \) in \( e' \)
7. \(\text{halt}^{\tau''}\)

Lemma 1.7 (Cannonical Forms) If \( \Delta; \Gamma \vdash v: \tau \) then one of the following must be true.

1. \( \tau = \text{unit} \) iff \( v = \langle \rangle \)
2. \( \tau = \tau_1^{\Delta''} \tau_2 \) iff \( v = (\lambda x: \tau_1.e)^{\Delta''} \) and \( \Delta = \Delta' \uplus \Delta'' \)
3. \( \tau = \forall \rho.\tau' \) iff \( v = (\Lambda \rho.e) \)
4. \( \tau = (\tau' \text{ at } \rho) \) iff \( v = \text{put}[\rho](v') \) and \( \Delta = \Delta' \uplus \{ \rho \} \)

Lemma 1.8 (Typing Relation Preservers Well Formedness) If \( \Delta; \Gamma \vdash e : \tau \) then \( \Delta \vdash \tau \text{ wf} \).

Lemma 1.9 (Typing Under Region Variable Substitution) If \( \Delta \uplus \{ \rho' \} \uplus \{ \rho \}; \Gamma \vdash e : \tau \) then \( \Delta \uplus \{ \rho' \}; \Gamma[\rho'/\rho] \vdash e[\rho'/\rho] : \tau[\rho'/\rho] \).

Lemma 1.10 (Control Context Independence) If \( \Delta; \Gamma \vdash E[e] : \tau \) then \( \Delta; \Gamma \vdash e : \tau' \).

Lemma 1.11 (Control Context Replacement) If \( \Delta; \Gamma \vdash e_1 : \tau, \Delta; \Gamma \vdash e_2 : \tau, \) and \( \Delta; \Gamma \vdash E[e_1] : \tau' \) then \( \Delta; \Gamma \vdash E[e_2] : \tau' \).

Lemma 1.12 (Region Stack Independence) If \( \Delta; \Gamma \vdash R[e] : \tau \) then there exists \( \Delta' \) such that \( \Delta \uplus \Delta'; \Gamma \vdash e : \tau \).

Lemma 1.13 (Region Stack Replacement) There exists \( \Delta' \) such that if \( \Delta \uplus \Delta'; \Gamma \vdash e_1 : \tau, \Delta \uplus \Delta'; \Gamma \vdash e_2 : \tau, \) and \( \Delta; \Gamma \vdash R[e_1] : \tau \) then \( \Delta; \Gamma \vdash R[e_2] : \tau \).
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 8: Iterative List Reverse

2.6 An Extended First-Order Variant

We can easily extend our core calculus to include sums, products, and recursive types. Our core calculus is higher-order but as we will see the higher-order features of our calculus are not strictly needed in order to implement a type-preserving garbage collector, and in fact make writing such a collector impossible since the garbage collector cannot copy values captured by the closure of a higher-order function. However, the safety of our higher-order calculus immediately implies the safety of a first-order variant, and the proof techniques for the higher-order calculus are simpler because there are fewer syntactic categories. The higher-order calculus is also interesting by itself and has applications beyond our type-preserving garbage collector. In the next section we will use a first-order variant extended into a full ML-like language, and we will adopt more traditional ML syntax in our discussion.

3 Example: itrev

Source program. Figure 8 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 that 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 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, 25, 32].

We will convert the program in Figure 8 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 9 illustrates Tolmach’s closure conversion technique. Notice that the types of any free variables are captured in the type of the closure [29].

Figure 10 is the result of applying 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. This type contains one data constructor Ret_r1 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. All functions have a return type of Ans, which means they do not return. 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 [13]. 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.3 However, providing true separate compilation using standard higher-order techniques that preserve abstraction and have better separate compilations properties

3Tolmach outlines a separate compilation technique that requires special support from the linker.
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 11 shows our program with explicit region annotations. Notice that the
type \( \text{lst} \) in figure 10 becomes a type constructor \( \text{lst}[\rho] \) 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 \( \text{Cons} \) data-constructor; this is reflected in the type
\( \text{Cons}(\text{int}, \text{lst}[\rho]) \) at \( \rho \).

Both the \( \text{itrev} \) and \( \text{apply} \) functions each take a single region parameter \( (\rho_{\text{alloc}}) \), which
corresponds to the allocation pointer in a normal untyped system. When we allocate a new
list cell we use the notation \( \text{lst}[\rho_{\text{heap}}].\text{Cons}(1, \ldots) \) which instantiate the region parameter
\( (\rho) \) of the type constructor \( \text{lst} \) to \( \rho_{\text{heap}} \) and indicates that the new list cell will be allocated
in the region \( \rho_{\text{heap}} \). We have assigned regions to types so that values are allocated in one
global region, which acts like a traditional heap. When we call \( \text{itrev} \) we instantiate its
region parameter \( \rho_{\text{alloc}} \) to \( \rho_{\text{heap}} \). We could apply a more refined local region 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 con-
tinuation. This approach simplifies the compilation of advanced control features such as
exceptions and first class continuations as well as simplifying the reasoning of safety. How-
ever, heap-allocating return continuations could impact performance in an undesirable way.
A system extended with linear types, along with a set of simple syntactic restriction would
allow us to stack allocate return continuations.
Figure 11: Program itrev after Region Annotation

**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 [13]. 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 12 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 identify where the error was by examining the code statically. 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.

**A Safe Flip.** Figure 13 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 ... requires that the body of the expression does not return, i.e. has type Ans, and can be safely evaluated using only
type lst[ρ] = Nil | Cons(int, 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 12: 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[ρalloc](...):Ans = ... and apply[ρalloc](...):Ans = ...

and gc[ρfrom](...):Ans = ...

and copy_gc_cont[ρfrom, ρto](x:gc_cont[ρfrom]):gc_cont[ρto] =

  case x of Ret_itrev(k, l, acc) ⇒
    let k' = copy_cont[ρfrom, ρto](k) in (* walk the "stack" *)
    let l' = copy_lst[ρfrom, ρto](l) in
    let acc' = copy_lst[ρfrom, ρto](acc)
    in gc_cont[ρto].Ret_itrev(k', l', acc')

and copy_lst[ρfrom, ρto](x:lst[ρfrom]):lst[ρto] = ...

and copy_cont[ρfrom, ρto](x:cont[ρfrom]):cont[ρto] = ...

...
Preserving Sharing. Consider the data structure in the first half of Figure 15. 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 16 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 17 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 \texttt{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 \texttt{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 \texttt{ref} constructor.
prim objId : [α] → int
tycon tbl :: Rgn → Typ → Typ → Typ = …
fun newTbl[ρtbl, α, β](sz:int):tbl[ρtbl, α, β] = …
fun inTbl[ρtbl, α, β](t:tbl[ρtbl, α, β], key:α):bool = …
fun getTbl[ρtbl, α, β](t:tbl[ρtbl, α, β], key:α):β = …
fun addTbl[ρtbl, α, β](t:tbl[ρtbl, α, β], key:α, val:β):unit = …

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

fun itrev[ρalloc](...) : Ans = …

... and share_copy_lst[ρtbl, ρfrom, ρto]
(t:tbl[ρtbl, lst[ρfrom], lst[ρto]], x:lst[ρfrom]):lst[ρto] = …

Figure 16: Preserving Sharing with a Hash-Table
tycon ref :: Rgn → Typ → Typ

type gc_lst[ρfrom, ρto] = Nil
  | Cons(ref[ρfrom,fwd_ptr[ρto]], int, gc_lst[ρfrom, ρto]) at ρfrom
and fwd_ptr[ρto] = NULL | PTR(lst[ρto])
and lst[ρto] = Nil
  | Cons(ref[ρto, fwd_null], int, lst[ρto]) at ρto
and fwd_null = NULL

fun itrev[ρalloc](...):Ans = ...

and share_copy_lst[ρfrom, ρto](x:gc_lst[ρfrom, ρto]):lst[ρto] =
  case x of Nil ⇒ lst[ρto].Nil
  | Cons(f, hd, tl) ⇒
    (case deref[ρfrom](f) of NULL ⇒
      let hd' = hd in
      let tl' = share_copy_lst[ρfrom, ρto] in
      let l = lst[ρto].Cons(mkref[ρto](fwd_null.NULL), hd', tl')
      in f := l; l
      PTR(l) ⇒ l)

Figure 17: Encoding Forwarding Pointers

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 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

5A ref cell can be thought of as a one element array.
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, et al [11] 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 Appendix B 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 δ−main encoding technique [13] 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 [16] 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 \((n = 1000)\) [28]

**inline** Inline variant of appel1 \((n = 1000)\) [28]

**appel2** Program designed to demonstrate issues of space complexity \((n = 1000)\) [28]

**ackermann** Ackermann’s function evaluated \((n = 3, m = 6)\) [28]

**fib** Recursive Fibonacci \((n = 33)\)

**hsum** Sum the value in a heap allocated list \((n = 1000)\) [28]

**quicksort** Quicksort randomly generated list \((n = 1000)\) [28]

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

**sum** Recursive sum of the first \(n\) integers \((n = 1000)\) [28]

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.

**Compiler.** We have modified the back end of the MLton [19] 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 hash-table (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 18 shows the total wall-clock run time for each program and variant normalized by the performance of the optimized CPS-converted program. Immediately, one can see that the CPS conversion can cause more than a factor of two performance degradation when compared to the original program, which is stack allocating activation records. We are using a simple flat-closure representation; more advanced closure representation techniques can significantly reduce the amount of allocation.[2, 3] Figure 19
shows that our CPS converted programs are allocating significantly more heap data\textsuperscript{6}, 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.\textsuperscript{7}

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 20 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.\textsuperscript{8} In this case

\textsuperscript{6}Notice that some programs did not allocate any heap data originally.

\textsuperscript{7}The extra auxiliary space need for garbage collection is not accounted for in the figure.

\textsuperscript{8}In the case of share-copy, which is allocating more data, because of our dynamic heap resizing policy
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 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 21 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.

it is being invoked at different times when there is less live data to be collected.
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 regions 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 [25, 32] with our system. [20] 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 [21] 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 man-
agement: 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.

heap cost for languages with closures. Journal of Functional Programming, 6(1):47–74,
January 1996.


October 1993.

[7] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Region analysis and the polymor-
phic lambda calculus. In Proceedings, Fourteenth Annual IEEE Symposium on Logic in
Press.

Conference on Programming Languages Design and Implementation, ACM SIGPLAN

[9] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equal-

[10] Karl Crary. Typed compilation of inclusive subtyping. In Proceedings of Fifth In-
ternational Conference on Functional Programming, pages 68–81, Montreal, Canada,
September 2000.

lus of capabilities. In Conference Record of the Twenty-sixth Annual ACM Symposium


A Proof of Safety Properties

Theorem 1 (Type Soundness) If \( \vdash P_1 \text{ wt} \) then, there is no stuck \( P_2 \) such that \( P_1 \mapsto_p^* P_2 \).

Proof. By structural induction on derivations of \( P \mapsto_p^* P' \) and Lemma 1.1 (Type Preservation of Programs) and Lemma 1.2 (Progress)

Induction Hypothesis: We need the following stronger induction hypothesis: If \( \vdash P_1 \text{ wt} \) and \( P_1 \mapsto_p^* P_2 \) then \( P_2 \) is not stuck and \( \vdash P_2 \text{ wt} \).

Lemma 1.1 (Type Preservation of Programs) If \( \vdash P_1 \text{ wt} \) and \( P_1 \mapsto_p^* P_2 \), then \( \vdash P_2 \text{ wt} \).

Proof. By case analysis of \( P_1 \mapsto_p^* P_2 \)

Consdier the cases of \( P_1 \mapsto_p^* P_2 \)

Lemma 1.2 (Progress)
4. \{\} \cup \Delta'; \{\} \vdash E[e_1]: \tau 
   \text{By Lemma 1.12 and (3)}
5. \{\} \cup \Delta'; \{\} \vdash e_1: \tau' 
   \text{By Lemma 1.10 and (5)}
6. \{\} \cup \Delta'; \{\} \vdash e_2: \tau' 
   \text{By Lemma 1.5 with (5) and (2)}
7. \{\} \cup \Delta'; \{\} \vdash E[e_2]: \tau 
   \text{By Lemma 1.11 with (5), (6), and (4)}
8. \{\}; \{\} \vdash R[E[e_2]]: \tau 
   \text{By Lemma 1.13 with (4), (7), and (3)}
   \text{Therefore} \vdash R[E[e_2]] \text{ wt} 
   \text{By wte and (8)}

\text{case rdsltr } R[E[\text{letr } \rho \text{ in } e]] \hookrightarrow_{p} R[\text{letr } \rho \text{ in } E[e]]

1. \vdash R[E[\text{letr } \rho \text{ in } e]] \text{ wt} 
   \text{By assumption}
2. \{\}; \{\} \vdash R[E[\text{letr } \rho \text{ in } e]]: \tau 
   \text{By (1) and inversion of wte}
3. \{\} \cup \Delta'; \{\} \vdash E[\text{letr } \rho \text{ in } e]: \tau 
   \text{By Lemma 1.12 and (2)}
4. \{\} \cup \Delta'; \{\} \vdash \text{letr } \rho \text{ in } e: \tau' 
   \text{By Lemma 1.10 and (3)}
5. \{\} \cup \Delta' \cup \{\rho\}; \{\} \vdash e: \tau' 
   \text{By (4) and inversion of htletr}
6. \{\} \cup \Delta' \cup \{\rho\}; \{\} \vdash \text{letr } \rho \text{ in } e: \tau' 
   \text{By (4) and weakening}
7. \{\} \cup \Delta' \cup \{\rho\}; \{\} \vdash E[\text{letr } \rho \text{ in } e]: \tau 
   \text{By (3) and weakening}
8. \{\} \cup \Delta' \cup \{\rho\}; \{\} \vdash E[e]: \tau 
   \text{By Lemma 1.11 with (6), (5), and (7)}
9. \{\} \cup \Delta' \vdash \tau \text{ wf} 
   \text{By (3) and Lemma 1.8}
10. \{\} \cup \Delta'; \{\} \vdash \text{letr } \rho \text{ in } E[e]: \tau 
    \text{By htletr with (9) and (8)}
11. \{\}; \{\} \vdash R[\text{letr } \rho \text{ in } E[e]]: \tau 
    \text{By Lemma 1.13 with (3), (10), and (2)}
   \text{Therefore} \vdash R[\text{letr } \rho \text{ in } E[e]] \text{ wt} 
   \text{By wte and (11)}

\text{case rdsgt } R[\text{letr } \rho \text{ in } R'[E[\text{get[}\rho\text{](put[}\rho\text{](v))]]]] \hookrightarrow_{p} R[\text{letr } \rho \text{ in } R'[E[v]]]

1. \vdash R[\text{letr } \rho \text{ in } R'[E[\text{get[}\rho\text{](put[}\rho\text{](v))]]]] \text{ wt} 
   \text{By assumption}
2. \{\}; \{\} \vdash R[\text{letr } \rho \text{ in } R'[E[\text{get[}\rho\text{](put[}\rho\text{](v))]]]]: \tau 
   \text{By (1) and inversion of wte}
3. \{\} \cup \Delta'; \{\} \vdash E[\text{get[}\rho\text{](put[}\rho\text{](v))]]: \tau 
   \text{By Lemma 1.12 and (2)}
4. \{\} \cup \Delta'; \{\} \vdash \text{get[}\rho\text{](put[}\rho\text{](v))]: \tau' 
   \text{By Lemma 1.10 and (3)}
5. \{\} \cup \Delta'; \{\} \vdash \text{put[}\rho\text{](v)]: (\tau' \text{ at } \rho) 
   \text{By (4) and inversion of htpget}
6. \{\} \cup \Delta'; \{\} \vdash v: \tau' 
   \text{By (5) and inversion of htput}
7. \{\} \cup \Delta'; \{\} \vdash E[v]: \tau 
   \text{By Lemma 1.11 with (4), (6), and (3)}
8. \{\}; \{\} \vdash R[\text{letr } \rho \text{ in } E[v]]: \tau 
   \text{By Lemma 1.13 with (3), (7), and (2)}
   \text{Therefore} \vdash R[\text{letr } \rho \text{ in } E[v]] \text{ wt} 
   \text{By wte and (8)}

\text{case rdsonly } R[E[\text{only } \Delta'' \text{ in } e]] \hookrightarrow_{p} R'\Delta''[e]

1. \vdash R[E[\text{only } \Delta'' \text{ in } e]] \text{ wt} 
   \text{By assumption}
2. \{\}; \{\} \vdash R[E[\text{only } \Delta'' \text{ in } e]] : \tau  
\quad \text{By (1) and inversion of wte}

3. \{\} \uplus \Delta; \{\} \vdash E[\text{only } \Delta'' \text{ in } e] : \tau  
\quad \text{By Lemma 1.12 with (2)}

4. \{\} \uplus \Delta; \{\} \vdash \text{only } \Delta'' \text{ in } e : \tau'  
\quad \text{By Lemma 1.10 with (3)}

5. \{\} \uplus \Delta' \uplus \Delta''; \{\} \vdash \text{only } \Delta'' \text{ in } e : \text{Ans}  
\quad \text{Because by inspection of } h\text{t}_\text{only } \Delta = \Delta' \uplus \Delta''  
\quad \text{By (5) inversion of } h\text{t}_\text{only}

6. \Delta''; \{\} \vdash e : \text{Ans}  
\quad \text{By definition of } R''_\Delta  
\quad \text{By wte with (7)}

7. \{\}; \{\} \vdash R[\Delta''[e]] \vdash \text{Ans}  
\quad \text{By (5) inversion of } h\text{t}_\text{only}

\textbf{Lemma 1.2 (Progress)} \textit{If } \vdash P_1 \text{ wt then, there exists } P_2 \text{ such that } P_1 \rightarrow P_2 \text{ or } P_2 \text{ is an answer. i.e. } P_1 \text{ is not stuck.}

\textbf{Proof.} \textit{Because } \vdash P_1 \text{ wt impiles } \{\}; \{\} \vdash R[e] : \tau \text{. By Lemma 1.12 } \Delta; \{\} \vdash e : \tau \text{ so by case analysis on the conclusions of Lemma 1.6}

\textbf{Consider the cases of Lemma 1.6}

\textbf{case } R\text{dshalt } R[E[\text{halt}^\tau]] \rightarrow P \text{ halt}^\tau \text{ where } E \neq []

1. \vdash R[E[\text{halt}^\tau]] \text{ wt}  
\quad \text{By assumption}

2. \{\}; \{\} \vdash R[E[\text{halt}^\tau]] : \tau  
\quad \text{By (1) and inversion of wte}

3. \{\} \vdash \tau \text{ wt}  
\quad \text{By Lemma 1.8 with (2)}

4. \{\}; \{\} \vdash \text{halt}^\tau : \tau  
\quad \text{By hthalt with (3)}

\text{Therefore } \vdash \text{halt}^\tau \text{ wt}  
\quad \text{By wte with (4)}

\textbf{case } R\text{dsfreer } R[\text{let } \rho \in R'[e]] \rightarrow P R[R'[e]] \text{ where } \vdash R[R'[e]] \text{ wt}  
\quad \text{Trivial since by assumption } \vdash R[R'[e]] \text{ wt}

\textbf{Lemma 1.2 (Progress)} \textit{If } \vdash P_1 \text{ wt then, there exists } P_2 \text{ such that } P_1 \rightarrow P_2 \text{ or } P_2 \text{ is an answer. i.e. } P_1 \text{ is not stuck.}

\textbf{Proof.} \textit{Because } \vdash P_1 \text{ wt impiles } \{\}; \{\} \vdash R[e] : \tau \text{. By Lemma 1.12 } \Delta; \{\} \vdash e : \tau \text{ so by case analysis on the conclusions of Lemma 1.6}

\textbf{Consider the cases of Lemma 1.6}

\textbf{case } e = v \textbf{ If } e \text{ is a value than } P_1 \text{ is an answer.}

\textbf{case } e = E[r] \text{ where } r = ((\lambda x : \tau'.e')^\Delta' v) \textbf{ P}_1 \text{ reduces to } P_2 \text{ using } \text{rdspure} \text{ with } \text{rdsbetav}.

\textbf{case } e = E[r] \text{ where } r = ((\Delta.\rho.e')[\rho]) \textbf{ P}_1 \text{ reduces to } P_2 \text{ using } \text{rdspure} \text{ with } \text{rdstapp}.

\textbf{case } e = E[r] \text{ where } r = (\text{fix } f : \tau'.v') \textbf{ P}_1 \text{ reduces to } P_2 \text{ using } \text{rdspure} \text{ with } \text{rdspure}.

\textbf{case } e = E[r] \text{ where } r = \text{let } \rho \text{ in } e' \textbf{ P}_1 \text{ reduces to } P_2 \text{ using } \text{rdsletr}.

\textbf{case } e = E[r] \text{ where } r = \text{get}[\rho](\text{put}[\rho](v)) \textbf{ P}_1 \text{ reduces to } P_2 \text{ using } \text{rdsget}.

\textbf{case } e = E[r] \text{ where } r = \text{only } \Delta' \text{ in } e' \textbf{ P}_1 \text{ reduces to } P_2 \text{ using } \text{rdsonly}.
Because $e = E[r]$ where $r = \text{halt}^{\tau'}$. If $E \neq \text{[]}$ then $P_1$ reduces to $P_2$ using $rdsfix$ with $rdsht$. If $E = \text{[]}$ then $P_1$ is an answer.

**Lemma 1.3 (Typing Under Term Substitution)** If $\Delta; \Gamma \vdash e : \tau$ and $\Delta; \Gamma \cup \{ x : \tau \} \vdash e' : \tau'$ then $\Delta; \Gamma \vdash e'[e/x] : \tau'$, where $e = v$ or $e = (\text{fixf} : \tau'' . v)$.

**Proof.** By structural induction on the derivations of $\Delta; \Gamma \cup \{ x : \tau \} \vdash e' : \tau'$.

**Induction Hypothesis:** If $\Delta; \Gamma \vdash e : \tau$ and $\Delta; \Gamma \cup \{ x : \tau \} \vdash e' : \tau'$ then $\Delta; \Gamma \vdash e'[e/x] : \tau'$, where $e = v$ or $e = (\text{fixf} : \tau'' . v)$.

**Case** 1: $\Delta; \Gamma \cup \{ x : \tau \} \vdash x : \tau$

Because $e' = x$, $\tau' = \tau$ and $x[e/x] = e$ therefore $\Delta; \Gamma \vdash e : \tau$ by assumption.

**Case** 2: $\Delta; \Gamma \cup \{ x : \tau \} \vdash y : \tau'$

Because $e' = y$, $y[e/x] = y$ therefore $\Delta; \Gamma \cup \{ y : \tau' \} \vdash y : \tau'$ by $\text{htvar}$.

**Case** 3: $\Delta; \Gamma \vdash \text{} : \text{unit}$

Because $e' = \langle \rangle$, $\tau' = \text{unit}$, and $\langle \rangle[e/x] = \langle \rangle$ therefore $\Delta; \Gamma \vdash \langle \rangle : \text{unit}$ by $\text{htunit}$.

By assumption and exchange

**Case** 4: $\Delta' \vdash (\Gamma' \cup \{ x : \tau \}) \text{wenv} \Delta'' \vdash \tau_1 \text{w} \Delta''' ; \Gamma'' \cup \{ y : \tau_1 \} \vdash e'' : \tau_2$

Because $\Gamma = \Delta' \cup \Delta'' ; (\Gamma' \cup \{ x : \tau \}) \cup \Gamma'' \vdash (\lambda y : \tau_1 . e'') \Delta'' ; \Delta'' ; (\Gamma' \cup \{ x : \tau \}) \cup \Gamma'' \vdash \Delta'' ; \Delta''$.

1. $\Delta'' ; \Gamma'' \cup \{ y : \tau_1 \} \vdash e'' : \tau_2$

2. $e''[e/x] = e''$

Therefore $\Delta' \cup \Delta'' ; \Gamma' \cup \Gamma'' \vdash (\lambda y : \tau_1 . e'') \Delta'' ; \Delta'' \vdash \Delta'' ; \Delta''$.

By assumption

**Case** 5: $\Delta' \vdash \Gamma' \text{wenv} \Delta'' \vdash \tau_1 \text{w} \Delta''' ; (\Gamma' \cup \{ x : \tau \}) \cup \Gamma'' \vdash e'' : \tau_2$

Because $\Gamma = \Delta' \cup \Delta'' ; \Gamma' \cup \Gamma'' \vdash (\lambda y : \tau_1 . e'') \Delta'' ; \Delta''$.

1. $\Delta' \cup \Delta'' ; \Gamma' \cup \Gamma'' \vdash e : \tau$

By assumption

2. $\Delta' \vdash \Gamma' \text{wenv}$

By assumption

3. $\Delta'' \vdash \tau_1 \text{w}$

By assumption

4. $\Delta'' ; \Gamma'' \cup \{ y : \tau_1 \} \cup \{ x : \tau \} \vdash e'' : \tau_2$

By assumption and exchange

5. $\Delta'' \vdash \Gamma'' \text{wenv}$

By assumption implicit in (4)
6. \( \Delta'' ; \Gamma'' \vdash e : \tau \)

By Lemma 1.4 with (3), (2), (5), and (1) N.B. here is where we use the fact that \( e = v \) or \( e = (\text{fix} f : \tau'.v') \)

7. \( \Delta'' ; \Gamma'' \cup \{ y : \tau_1 \} \vdash e''[e/x] : \tau_2 \)

Therefore \( \Delta' \cup \Delta'' ; \Gamma' \cup \Gamma'' \vdash (\lambda y : \tau_1. e''[e/x]) \Delta'' : \tau_1 \Delta'' \)

By \( \text{httabs} \) with (2), (3) and (7)

\[
\begin{array}{c}
\frac{
\Delta' \cup \Delta'' ; \Gamma \cup \{ x : \tau \} \vdash e_1 : \tau_1 \Delta'' \quad \Delta' \cup \Delta'' ; \Gamma \cup \{ x : \tau \} \vdash e_2 : \tau_1 
}{
\frac{
\Delta' \cup \Delta'' ; \Gamma \cup \{ x : \tau \} \vdash (e_1 e_2) : \tau_2 
}{

\Delta' \cup \Delta'', e' = (e_1 e_2), \tau' = \tau_2, \text{ and } (e_1 e_2)[e/x] = (e_1[e/x] e_2[e/x])
}\end{array}
\]

Because \( \Delta = \)

\[
\begin{array}{c}
\frac{
\Delta' \cup \Delta'' ; \Gamma \vdash e : \tau 
}{
\frac{
\Delta' \cup \Delta'' ; \Gamma \vdash e'' : \tau'' 
}{
\Delta ; \Gamma \vdash (\lambda \rho.e'') : \forall \rho.\tau'' 
}\end{array}
\]

Because \( e' = (\Lambda \rho. e''), \tau' = \forall \rho.\tau'' \), and

\[
\begin{array}{c}
\frac{
\Delta \vdash \lambda x.\tau 
}{
\frac{
\Delta \vdash \text{let} \rho \in e'' : \forall \rho.\tau'' 
}{
\Delta \vdash \text{let} \rho \in (e''[\rho]) : \tau''[\rho'/\rho] 
}\end{array}
\]

Because \( e' = (e''[\rho]) \), \( \tau' = \tau''[\rho'/\rho] \),

and \( (e''[\rho])[e/x] = (e''[e/x]) \rho' \)

Because \( e' = (e''[\rho']) \), \( \tau' = \tau''[\rho'/\rho] \),

and \( (e''[\rho'])[e/x] = (e''[e/x]) \rho' \)

Because \( e' = \text{let} \rho \in e'' \) and
1. $\Delta; \Gamma \vdash e : \tau$  
   By assumption
2. $\Delta \vdash \tau'$ wf  
   By assumption
3. $\Delta \cup \{\rho\}; \Gamma \cup \{x : \tau\} \vdash e'' : \tau'$  
   By IH with (1) and (3)

Therefore $\Delta; \Gamma \vdash \text{letr } \rho \text{ in } e''[e/x] : \tau'$  
   By letr with (2) and (4)

\[
\begin{array}{c}
\text{case} \\
\Delta' \cup \{\rho\}; \Gamma \cup \{x : \tau\} \vdash e' : \tau'' \\
\hline
\Delta' \cup \{\rho\}; \Gamma \cup \{x : \tau\} \vdash \text{put}[\rho](e'') : (\tau'' \text{ at } \rho) \\
\hline
\end{array}
\]

Because $\Delta = \Delta' \cup \{\rho\}$, $e' = \text{htput}$

\[
\begin{array}{c}
\text{case} \\
\Delta' \cup \{\rho\}; \Gamma \cup \{x : \tau\} \vdash e' : (\tau' \text{ at } \rho) \\
\hline
\Delta' \cup \{\rho\}; \Gamma \cup \{x : \tau\} \vdash \text{get}[\rho](e'') : \tau' \\
\hline
\end{array}
\]

Because $\Delta = \Delta' \cup \{\rho\}$, $e'' = \text{get}[\rho](e'')$

\[
\begin{array}{c}
\Delta \vdash \Gamma' \cup \{x : \tau\} \text{ wenv} \\
\hline
\Delta''; \Gamma'' \vdash e'' : \text{Ans} \\
\hline
\end{array}
\]

Because $\Delta = \Delta' \cup \Delta''$, $\Gamma = \Gamma' \cup \Gamma''$, $e' = \text{only } \Delta'' \text{ in } e''$ and $\tau' = \text{Ans}$

\[
\begin{array}{c}
\text{case} \\
\Delta' \cup \Delta'' \cup \{x : \tau\} \text{ wenv} \\
\hline
\Delta''; \Gamma'' \cup \{x : \tau\} \vdash \text{get}[\rho](e'') : \tau' \\
\hline
\end{array}
\]

Because $\Delta = \Delta' \cup \Delta'', \Gamma = \Gamma' \cup \Gamma''$, $e' = \text{only } \Delta'' \text{ in } e''$, only $\Delta'' \text{ in } e''[e/x] = \text{only } \Delta'' \text{ in } e''[e/x]$, and $\tau' = \text{Ans}$

\[
\begin{array}{c}
\text{case} \\
\Delta' \vdash \Gamma' \text{ wenv} \\
\hline
\Delta''; \Gamma'' \cup \{x : \tau\} \vdash e'' : \text{Ans} \\
\hline
\end{array}
\]

Because $\Delta = \Delta' \cup \Delta'', \Gamma = \Gamma' \cup \Gamma''$, $e' = \text{only } \Delta'' \text{ in } e''$, only $\Delta'' \text{ in } e''[e/x] = \text{only } \Delta'' \text{ in } e''[e/x]$, and $\tau' = \text{Ans}$

\[
\begin{array}{c}
\text{case} \\
\Delta' \cup \Delta'' ; \Gamma' \cup \Gamma'' \vdash e : \tau \\
\hline
\Delta''; \Gamma'' \cup \{x : \tau\} \vdash e'' : \text{Ans} \\
\hline
\end{array}
\]

By assumption
3. $\Delta' \vdash \Gamma' \text{ wfenv}$
4. $\Delta'' \vdash \Gamma'' \cup \{x:\tau\} \text{ wfenv}$
5. $\Delta'' \vdash \tau \text{ wfenv}$
6. $\Delta''; \Gamma'' \vdash e: \tau$
7. $\Delta''; \Gamma'' \vdash e'[e/x] : \text{Ans}$

Therefore $\Delta' \cup \Delta''; \Gamma' \cup \Delta'' \vdash \text{only } \Delta'' \text{ in } e''[e/x] : \text{Ans}$

Because $e' = (\text{fix} \ f : \tau'.v)$ and $(\text{fix} \ f : \tau'.v)[e/x]$.

<table>
<thead>
<tr>
<th>Case</th>
<th>$\Delta \vdash \tau' \text{ wf} \quad \Delta; \Gamma \cup {\tau : x} \cup {f : \tau'} \vdash v : \tau'$</th>
<th>$\Delta; \Gamma \cup {\tau : x} \vdash (\text{fix} \ f : \tau'.v) : \tau'$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\Delta; \Gamma \cup {\tau : x} \vdash \text{htfix}$</td>
<td>$\text{htfix}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case</th>
<th>$\Delta \vdash \tau' \text{ wfix} \quad \Delta; \Gamma \cup {\tau : x} \vdash \text{halt}'^{\tau'} $</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\Delta; \Gamma \cup {\tau : x} \vdash \text{htfix}$</td>
</tr>
</tbody>
</table>

$\Delta \vdash \tau'$ wfix therefore $\Delta; \Gamma \vdash \text{halt}'^{\tau'}$ by hthalt.

Lemma 1.4 (Region Context Strengthening) If $\Delta' \vdash \tau$ wfix, $\Delta \vdash \Gamma$ wfenv, $\Delta' \vdash \Gamma'$ wfenv, and $\Delta \cup \Delta' ; \Gamma \cup \Gamma' \vdash e : \tau$ where $e = v$ or $e = (\text{fix} f : \tau . v)$ then $\Delta' ; \Gamma' \vdash e : \tau$.

Proof. By induction on derivations of $\Delta \cup \Delta' ; \Gamma \cup \Gamma' \vdash e : \tau$.

Induction Hypothesis: If $\Delta' \vdash \tau$ wfix, $\Delta \vdash \Gamma$ wfenv, $\Delta' \vdash \Gamma'$ wfenv, and $\Delta \cup \Delta' ; \Gamma \cup \Gamma' \vdash e : \tau$ where $e = v$ or $e = (\text{fix} f : \tau . v)$ then $\Delta' ; \Gamma' \vdash e : \tau$.

<table>
<thead>
<tr>
<th>Case</th>
<th>$\Delta \cup \Delta' ; \Gamma \cup \Gamma' \vdash {} : \text{unit}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\text{htunit}$</td>
</tr>
</tbody>
</table>

Trivial since $\Delta' \vdash \Gamma'$ wfenv therefore $\Delta' ; \Gamma' \vdash \{\} : \text{unit}$ by hthalt.

<table>
<thead>
<tr>
<th>Case</th>
<th>$\Delta \vdash \Gamma \text{ wfenv} \quad \Delta' \vdash \tau_1 \text{ wfix} \quad \Delta' ; \Gamma \cup {x : \tau_1} \vdash e' : \tau_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\Delta \cup \Delta' ; \Gamma \cup \Gamma' \vdash (\lambda x : \tau_1.e')^{\Delta'} : \tau_1 \Rightarrow \tau_2 \text{htabs}$</td>
</tr>
</tbody>
</table>

Because $\tau = \tau_1 \Rightarrow \tau_2$.

1. $\Delta' \vdash \tau_1 \text{ wfix}$
2. $\Delta' ; \Gamma' \cup \{x : \tau_1\} \vdash e' : \tau_2$
3. $\{\} \vdash \{\} \text{ wfenv}$

Therefore $\{\} \cup \Delta' \cup \{\} \vdash (\lambda x : \tau_1.e')^{\Delta'} : \tau_1 \Rightarrow \tau_2$ by htabs with (3), (1), and (2).
Lemma 1.5 (Type Preservation of Expression)  If \( \Delta; \{ \} \vdash e_1 : \tau \) and \( e_1 \mapsto_e e_2 \), then \( \Delta; \{ \} \vdash e_2 : \tau \).
Proof. By case analysis of $e_1 \mapsto_e e_2$

Consider the cases of $e_1 \mapsto_e e_2$

**case** `rdsbetav` \[\((\lambda x: \tau_1.e)_{\Delta''} \mapsto_e e[v/x]\)\] Because $\Delta = \Delta' \sqcup \Delta''$ and $\{\} = \{\} \sqcup \{\}$

1. $\Delta' \cup \Delta''; \{\} \sqcup \{\} \vdash ((\lambda x: \tau_1.e)_{\Delta''} \mapsto_e e[v/x]: \tau_2)$
   By assumption
2. $\Delta' \cup \Delta''; \{\} \sqcup \{\} \vdash (\lambda x: \tau_1.e)_{\Delta''} \Delta'_1 \mapsto_e \tau_2$
   By (1) and inversion of `htapp`
3. $\Delta' \cup \Delta''; \{\} \sqcup \{\} \vdash v: \tau_1$
   By (1) and inversion of `htapp`
4. $\Delta''; \{\} \sqcup \{x: \tau_1\} \vdash e: \tau_2$
   By (2) and inversion of `httabs`
5. $\Delta' \cup \Delta''; \{\} \sqcup \{\} \sqcup \{x: \tau_1\} \vdash e: \tau_2$
   By (4) and weakening

   It follows that $\Delta' \cup \Delta''; \{\} \sqcup \{\} \vdash e[v/x]: \tau_2$
   By Lemma 1.3 with (3) and (5)

**case** `rdstapp` \[\((\Lambda \rho.e)[\rho']\) \mapsto_e e[\rho'/\rho]\] Because $\Delta = \Delta' \cup \{\rho'\}$

1. $\Delta' \cup \{\rho'\}; \{\} \vdash (\Lambda \rho.e)[\rho']:\tau'[\rho'/\rho]$
   By assumption
2. $\Delta' \cup \{\rho'\}; \{\} \vdash (\Lambda \rho.e): \forall \rho. \tau'$
   By (1) and inversion of `htapp`
3. $\Delta' \cup \{\rho'\} \cup \{\rho\}; \{\} \vdash e: \tau'$
   By (2) and inversion of `httabs`
4. $\Delta' \cup \{\rho'\}; \{\} \sqcup \{\rho'/\rho\} \vdash e[\rho'/\rho]: \tau'[\rho'/\rho]$
   By Lemma 1.9 with (2)

   It follows that $\Delta' \cup \{\rho'\}; \{\} \vdash e[\rho'/\rho]: \tau'[\rho'/\rho]$
   Because $\{\}[\rho'/\rho] = \{\}$

**case** `rdsfix` \[(\text{fix} f: \tau.v) \mapsto_e e[(\text{fix} f: \tau.v)/f]\]  

1. $\Delta; \{\} \vdash (\text{fix} f: \tau.v) : \tau$
   By assumption
2. $\Delta; \{\} \cup \{f: \tau\} \vdash v: \tau$
   By (1) and inversion of `htfix`

   It follows that $\Delta; \{\} \vdash v[(\text{fix} f: \tau.v)/f] : \tau$
   By Lemma 1.3 with (1) and (2)

**Lemma 1.6 (Redux Decomposition)** If $\Delta; \{\} \vdash e: \tau$ then $e$ is a value or $e = E[r]$ where $r$ is a redux. A redux is any of the following forms:

1. \[((\lambda x: \tau'.e')_{\Delta''} \mapsto_e \)\]
2. ```(\Lambda \rho.e')[\tau']```
3. `fix f: \tau'.v`
4. `letr \rho in e'`
5. `get[\rho](\text{put}[\rho](e'))`
6. `only \Delta' in e'`
7. `halt'`
Proof. By structural induction on well typed closed $e$

_Induction Hypothesis:_ If $\Delta; \{\} \vdash e : \tau$ then $e$ is a value or $e = E[r]$ where $r$ is a redux.

case $e = \emptyset$ $e$ is a value

case $e = (\langle \lambda x : \tau'.e' \rangle)^{\Delta''}$ where $\Delta = \Delta' \uplus \Delta''$ $e$ is a value

case $e = (\Lambda \rho.e')$ $e$ is a value

case $e = \text{put} [\rho](v)$ $e$ is a value

case $e = (v_1 v_2)$ Since $e$ is well typed by inversion of $\text{htapp}$ $v_1$ has type $\tau_1 \xrightarrow{\Delta''} \tau_2$ where $\Delta = \Delta' \uplus \Delta''$. From Lemma 1.7 we conclude that $v_1 = (\langle \lambda x : \tau'.e' \rangle)^{\Delta''}$. Therefore $E = [\ ]$ and $r = ((\langle \lambda x : \tau'.e' \rangle)^{\Delta''} v_2)$.

case $e = (v e')$ By IH $e' = E'[r']$. Therefore $E = (v E')$ and $r = r'$

case $e = (e' e'')$ By IH $e' = E'[r']$. Therefore $E = (E' e'')$ and $r = r'$

case $e = (v[\tau'])$ Since $e$ is well typed by inversion of $\text{htapp} v$ has type $\forall \rho. \tau'$. From Lemma 1.7 we conclude that $v = (\langle \Lambda \rho.e' \rangle)$. Therefore $E = [\ ]$ and $r = ((\langle \Lambda \rho.e' \rangle)[\tau'])$

case $e = (e[\tau'])$ By IH $e = E'[r']$. Therefore $E = (E[\tau'])$ and $r = r'$

case $e = \text{letr} \rho$ in $e'$ Let $E = [\ ]$ and $r = \text{letr} \rho$ in $e'$

case $e = \text{put} [\rho](e')$ By IH $e' = E'[r']$. Therefore $E = \text{put} [\rho](E')$ and $r = r'$

case $e = \text{get} [\rho](v)$ Since $e$ is well typed by inversion of $\text{htget} v$ has type $(\tau'$ at $\rho$) From 1.7 we conclude that $v = \text{put} [\rho](v')$. Therefore $E = [\ ]$ and $r = \text{get} [\rho](\text{put} [\rho](v'))$

case $e = \text{get} [\rho](e')$ By IH $e' = E'[r']$. Therefore $E = \text{get} [\rho](E')$ and $r = r'$

case $e = \text{only} \Delta'$ in $e'$ Let $E = [\ ]$ and $r = \text{only} \Delta'$ in $e'$

case $e = (\text{fix} f : \tau'.v')$ Let $E = [\ ]$ and $r = (\text{fix} f : \tau'.v')$

case $e = \text{halt}^\tau$ Let $E = [\ ]$ and $r = \text{halt}^\tau$

**Lemma 1.7 (Canonical Forms)** If $\Delta; \Gamma \vdash v : \tau$ then one of the following must be true.

1. $\tau = \text{unit}$ iff $v = \emptyset$

2. $\tau = \tau_1 \xrightarrow{\Delta''} \tau_2$ iff $v = (\langle \lambda x : \tau_1.e \rangle)^{\Delta''}$ and $\Delta = \Delta' \uplus \Delta''$

3. $\tau = \forall \rho. \tau'$ iff $v = (\Lambda \rho.e)$

4. $\tau = (\tau'$ at $\rho$) iff $v = \text{put} [\rho](v')$ and $\Delta = \Delta' \uplus \{\rho\}$
Proof. By inspection of the typing judgments for $\tau$

case $[\tau = \text{unit}]$ Follows from inversion of $\text{htunit}$

case $[\tau = \tau_1 \xrightarrow{\Delta''} \tau_2]$ Follows from inversion of $\text{htabs}$

case $[\tau = \forall \rho. \tau']$ Follows from inversion of $\text{httabs}$

case $[\tau = (\tau' \text{ at } \rho)]$ Follows from inversion of $\text{htput}$

Lemma 1.8 (Typing Relation Preservers Well Formedness) If $\Delta; \Gamma \vdash e : \tau$ then $\Delta \vdash \tau$ wf.

Proof. By structural induction on derivations of $\Delta; \Gamma \vdash e : \tau$.

Induction Hypothesis: If $\Delta; \Gamma \vdash e : \tau$ then $\Delta \vdash \tau$ wf.

- case $\Delta; \Gamma \uplus \{x : \tau\} \vdash x : \tau$
  - $\Delta \vdash \Gamma \uplus \{x : \tau\}$ wf by implicit assumption. Therefore $\Delta \vdash \tau$ wf by $\Delta \vdash \Gamma \uplus \{x : \tau\}$ wf and inversion of $\text{wfenvbv}$

- case $\Delta; \Gamma \vdash \langle \rangle : \text{unit}$
  - $\Delta \vdash \text{unit}$ wf by $\text{wfunit}$

- case $\Delta' \vdash \Gamma'$ $\text{wfenv}$ $\Delta'' \vdash \tau_1$ wf $\Delta''; \Gamma'' \uplus \{x : \tau_1\} \vdash e' : \tau_2$
  - $\Delta' \uplus \Delta''; \Gamma' \uplus \Gamma'' \vdash (\lambda x : \tau_1. e')^{\Delta''} : \tau_1 \xrightarrow{\Delta''} \tau_2$
  - Because $\Delta = \Delta' \uplus \Delta''$ and $\Gamma = \Gamma' \uplus \Gamma''$
    1. $\Delta'' \vdash \tau_1$ wf By assumption
    2. $\Delta''; \Gamma'' \uplus \{x : \tau_1\} \vdash e' : \tau_2$ By assumption
    3. $\Delta'' \vdash \tau_2$ wf By IH with (2)
    4. $\Delta'' \vdash \tau_1 \xrightarrow{\Delta''} \tau_2$ wf By $\text{wfarrows}$ with (1) and (3)
    Therefore $\Delta' \uplus \Delta'' \vdash \tau_1 \xrightarrow{\Delta''} \tau_2$ wf By weakening

- case $\Delta' \uplus \Delta''; \Gamma \vdash e_1 : \tau_1 \xrightarrow{\Delta''} \tau_2$ $\Delta' \uplus \Delta''; \Gamma \vdash e_2 : \tau_1$
  - $\Delta' \uplus \Delta''; \Gamma \vdash (e_1, e_2) : \tau_2$
  - Because $\Delta = \Delta' \uplus \Delta''$ and $\Gamma = \Gamma' \uplus \Gamma''$
    1. $\Delta' \uplus \Delta''; \Gamma' \uplus \Gamma'' \vdash e_1 : \tau_1 \xrightarrow{\Delta'} \tau_2$
    2. $\Delta' \uplus \Delta'' \vdash \tau_1 \xrightarrow{\Delta''} \tau_2$ wf By IH with (1)
    Therefore $\Delta' \uplus \Delta'' \vdash \tau_2$ wf By (2) and inversion of $\text{wfarrows}$
Because $\Delta = \Delta$

**case**

\[
\begin{align*}
\frac{\Delta \cup \{ \rho \}; \Gamma \vdash e' : \tau'}{\Delta; \Gamma \vdash (\lambda \rho.e') : \forall \rho.\tau'} \quad \text{httabs}
\end{align*}
\]

1. $\Delta \cup \{ \rho \}; \Gamma \vdash e' : \tau'$ By assumption
2. $\Delta \cup \{ \rho \} \vdash \tau' \text{ wf}$ By IH with (1)
3. $\therefore \Delta \vdash \forall \rho.\tau' \text{ wf}$ By $\text{wf}$ with (2)

**case**

\[
\begin{align*}
\frac{\Delta' \cup \{ \rho' \}; \Gamma \vdash \forall e' : \forall \rho.\tau'}{\Delta' \cup \{ \rho' \}; \Gamma \vdash (\rho'[\rho]) : \tau'[\rho/e']} \quad \text{htapp}
\end{align*}
\]

1. $\Delta' \cup \{ \rho' \}; \Gamma \vdash e' : \forall \rho.\tau'$ By assumption
2. $\Delta' \cup \{ \rho' \} \vdash \forall \rho.\tau' \text{ wf}$ By IH with (1)
3. $\Delta' \cup \{ \rho' \} \cup \{ \rho \} \vdash \tau' \text{ wf}$ By (2) and inversion of $\text{wfall}$
4. $\therefore \Delta' \cup \{ \rho' \} \vdash \tau'[\rho/\rho] \text{ wf}$ By induction on the derivations of $\Delta' \cup \{ \rho' \} \cup \{ \rho \} \vdash \tau' \text{ wf}$ and (3)

**case**

\[
\begin{align*}
\frac{\Delta \vdash \tau \text{ wf} \quad \Delta \cup \{ \rho \}; \Gamma \vdash e' : \tau}{\Delta; \Gamma \vdash (\text{let } \rho \text{ in } e') : \tau} \quad \text{htlet}
\end{align*}
\]

$\Delta \vdash \tau \text{ wf}$ by assumption

**case**

\[
\begin{align*}
\frac{\Delta' \cup \{ \rho \}; \Gamma \vdash e' : \tau'}{\Delta' \cup \{ \rho \}; \Gamma \vdash \text{put}[\rho](e') : (\tau' \text{ at } \rho)} \quad \text{htput}
\end{align*}
\]

Because $\Delta = \Delta' \cup \{ \rho \}$

1. $\Delta' \cup \{ \rho \} \vdash e' : \tau'$ By assumption
2. $\Delta' \cup \{ \rho \} \vdash \tau' \text{ wf}$ By IH with (1)
3. $\therefore \Delta' \cup \{ \rho \} \vdash (\tau' \text{ at } \rho) \text{ wf}$ By $\text{wfat}$ with (2)

**case**

\[
\begin{align*}
\frac{\Delta' \cup \{ \rho \}; \Gamma \vdash e' : (\tau' \text{ at } \rho)}{\Delta' \cup \{ \rho \}; \Gamma \vdash \text{get}[\rho](e') : \tau'} \quad \text{htget}
\end{align*}
\]

Because $\Delta = \Delta' \cup \{ \rho \}$

1. $\Delta' \cup \{ \rho \} \vdash e' : (\tau' \text{ at } \rho)$ By assumption
2. $\Delta' \cup \{ \rho \} \vdash (\tau' \text{ at } \rho) \text{ wf}$ By IH with (1)
3. $\therefore \Delta' \cup \{ \rho \} \vdash \tau' \text{ wf}$ By (2) and inversion of $\text{wfat}$

**case**

\[
\begin{align*}
\frac{\Delta' \vdash \Gamma \text{ wienv} \quad \Delta''; \Gamma'' \vdash e' \text{: Ans}}{\Delta' \cup \Delta''; \Gamma' \cup \Gamma'' \vdash (\text{only } \Delta'' \text{ in } e') \text{: Ans}} \quad \text{htonly}
\end{align*}
\]

Because $\Delta = \Delta' \cup \Delta''$, $\Delta' \cup \Delta'' \vdash \text{Ans} \text{ wf}$ by $\text{wfa}$

**case**

\[
\begin{align*}
\frac{\Delta \vdash \tau \text{ wf} \quad \Delta; \Gamma \cup \{ f : \tau \} \vdash v : \tau}{\Delta; \Gamma \vdash (\text{fix } f : \tau.v) : \tau} \quad \text{htfix}
\end{align*}
\]

$\Delta \vdash \tau \text{ wf}$ by assumption
Lemma 1.9 (Typing Under Region Variable Substitution) If $\Delta \uplus \{\rho\} \uplus \{\rho\}; \Gamma \vdash e : \tau$ then $\Delta \uplus \{\rho\}; \Gamma[\rho'/\rho] \vdash e[\rho'/\rho] : \tau[\rho'/\rho]$.

Proof. By structural induction on the derivations of $\Delta \uplus \{\rho\} \uplus \{\rho\}; \Gamma \vdash e : \tau$.

Induction Hypothesis: If $\Delta \uplus \{\rho\} \uplus \{\rho\}; \Gamma \vdash e : \tau$ then $\Delta \uplus \{\rho\}; \Gamma[\rho'/\rho] \vdash e[\rho'/\rho] : \tau[\rho'/\rho]$.

Lemma 1.10 (Control Context Independence) If $\Delta; \Gamma \vdash E[e] : \tau$ then $\Delta; \Gamma \vdash e : \tau'$.


Induction Hypothesis: If $\Delta; \Gamma \vdash E[e] : \tau$ then $\Delta; \Gamma \vdash e : \tau'$.

case $E = [\varepsilon]$ Because $E[e] = e$ and $\tau = \tau'$ therefore $\Delta; \Gamma \vdash e : \tau$ by assumption

case $E = (E' e')$ Because $E[e] = (E'[e] e')$ and $\Delta = \Delta' \uplus \Delta''$

1. $\Delta' \uplus \Delta''; \Gamma \vdash (E'[e] e') : \tau$ By assumption
2. $\Delta' \uplus \Delta''; \Gamma \vdash E'[e] : \tau_1 \Delta'' \Rightarrow \tau$ By (1) and inversion of $htapp$
   Therefore $\Delta' \uplus \Delta''; \Gamma \vdash e : \tau'$ By IH and (2)

case $E = (v E')$ Because $E[e] = (v E'[e])$ and $\Delta = \Delta' \uplus \Delta''$

1. $\Delta' \uplus \Delta''; \Gamma \vdash (v E'[e]) : \tau$ By assumption
2. $\Delta' \uplus \Delta''; \Gamma \vdash E'[e] : \tau_1$ By (1) and inversion of $htapp$
   Therefore $\Delta' \uplus \Delta''; \Gamma \vdash e : \tau'$ By IH and (2)

case $E = (E'[\rho'])$ Because $E[e] = (E'[e][\rho'])$

1. $\Delta; \Gamma \vdash (E'[e][\rho']) : \tau$ By assumption
2. $\Delta; \Gamma \vdash E'[e] : \forall \rho. \tau''$ By (1) and inversion of $htapp$
   Therefore $\Delta; \Gamma \vdash e : \tau'$ By IH and (2)

case $E = \text{put}[\rho](E')$ Because $E[e] = (\rho E'[e])$ and $\Delta = \Delta' \uplus \{\rho\}$

1. $\Delta' \uplus \{\rho\}; \Gamma \vdash \text{put}[\rho](E'[e]) : \tau$ By assumption
2. $\Delta' \uplus \{\rho\}; \Gamma \vdash E'[e] : \tau''$ By (1) and inversion of $htput$
   Therefore $\Delta' \uplus \{\rho\}; \Gamma \vdash e : \tau'$ By IH and (2)

case $E = \text{get}[\rho](E')$ Because $E[e] = (\rho E'[e])$ and $\Delta = \Delta' \uplus \{\rho\}$
1. $\Delta' \uplus \{\rho\}; \Gamma \vdash \text{get}[\rho](E'[e]):\tau$
   By assumption
2. $\Delta' \uplus \{\rho\}; \Gamma \vdash E'[e]:\tau''$
   By (1) and inversion of htget

Therefore $\Delta' \uplus \{\rho\}; \Gamma \vdash e:\tau'$
By IH and (2)

**Lemma 1.11 (Control Context Replacement)** If $\Delta; \Gamma \vdash e_1:\tau$, $\Delta; \Gamma \vdash e_2:\tau$, and $\Delta; \Gamma \vdash E[e_1]:\tau'$ then $\Delta; \Gamma \vdash E[e_2]:\tau'$.

**Proof.** By induction on the structure of $E$

*Induction Hypothesis:* If $\Delta; \Gamma \vdash e_1:\tau$, $\Delta; \Gamma \vdash e_2:\tau$, and $\Delta; \Gamma \vdash E[e_1]:\tau'$ then $\Delta; \Gamma \vdash E[e_2]:\tau'$.

**case** $E = [\ ]$ Because $E[e_2] = e_2$ and $\tau = \tau'$ therefore $\Delta; \Gamma \vdash e_2:\tau$ by assumption

**case** $E = (E' \ e')$ Because $E[e_1] = (E'[e_1] \ e')$ and $\Delta = \Delta' \uplus \Delta''$

1. $\Delta' \uplus \Delta''; \Gamma \vdash e_1:\tau$
   By assumption
2. $\Delta' \uplus \Delta''; \Gamma \vdash e_2:\tau$
   By assumption
3. $\Delta' \uplus \Delta''; \Gamma \vdash (E'[e_1] \ e') : \tau'$
   By assumption
4. $\Delta' \uplus \Delta''; \Gamma \vdash E'[e_1] : \tau_1 \overset{\Delta''}{\rightarrow} \tau'$
   By (3) and inversion of htapp
5. $\Delta' \uplus \Delta''; \Gamma \vdash e' : \tau_1$
   By (3) and inversion of htapp
6. $\Delta' \uplus \Delta''; \Gamma \vdash E'[e_2] : \tau_1 \overset{\Delta''}{\rightarrow} \tau'$
   By IH with (1), (2), and (4)

Therefore $\Delta' \uplus \Delta''; \Gamma \vdash (E'[e_2] \ e') : \tau'$
By htapp with (6) and (5)

**case** $E = (v \ E')$ Because $E[e_1] = (v \ E'[e_1])$ and and $\Delta = \Delta' \uplus \Delta''$

1. $\Delta' \uplus \Delta''; \Gamma \vdash e_1:\tau$
   By assumption
2. $\Delta' \uplus \Delta''; \Gamma \vdash e_2:\tau$
   By assumption
3. $\Delta' \uplus \Delta''; \Gamma \vdash (v \ E'[e_1]) : \tau'$
   By assumption
4. $\Delta' \uplus \Delta''; \Gamma \vdash v : \tau_1 \overset{\Delta''}{\rightarrow} \tau'$
   By (3) and inversion of htapp
5. $\Delta' \uplus \Delta''; \Gamma \vdash E'[e_1] : \tau_1$
   By (3) and inversion of htapp
6. $\Delta' \uplus \Delta''; \Gamma \vdash E'[e_2] : \tau'$
   By IH with (1), (2), and (5)

Therefore $\Delta' \uplus \Delta''; \Gamma \vdash (v \ E'[e_2]) : \tau'$
By htapp with (4) and (6)

**case** $E = (E'[\rho])$ Because $E[e_1] = (E'[e_1][\rho])$ and $\Delta = \Delta' \uplus \{\rho\}$

1. $\Delta' \uplus \{\rho\}; \Gamma \vdash e_1:\tau$
   By assumption
2. $\Delta' \uplus \{\rho\}; \Gamma \vdash e_2:\tau$
   By assumption
3. $\Delta' \uplus \{\rho\}; \Gamma \vdash (E'[e_1][\rho]) : \tau'$
   By assumption
4. $\Delta' \uplus \{\rho\}; \Gamma \vdash E'[e_1] : \forall \rho.\tau''$
   By (3) and inversion of htapp
5. $\Delta' \uplus \{\rho\}; \Gamma \vdash E'[e_2]: \forall \rho. \tau''$

Therefore $\Delta' \uplus \{\rho\}; \Gamma \vdash (E'[e_2][\rho']): \tau'$

By IH with (1), (2), and (4)

By httapp with (5)

**Lemma 1.12 (Region Stack Independence)** If $\Delta; \Gamma \vdash R[e]: \tau$ then there exists $\Delta'$ such that $\Delta \uplus \Delta'; \Gamma \vdash e: \tau$.

**Proof.** By induction on the structure of $R$

*Induction Hypothesis:* If $\Delta; \Gamma \vdash R[e]: \tau$ then there exists $\Delta'$ such that $\Delta \uplus \Delta'; \Gamma \vdash e: \tau$.

**Case $\overline{R = []}$** Because $R[e] = e$ and $\Delta = \Delta \uplus \{\}$ therefore $\Delta; \Gamma \vdash e: \tau$ by assumption

**Case $\overline{R = \text{let}\rho \text{ in } R'}$** Because $R[e] = \text{let}\rho \text{ in } R'[e]$

1. $\Delta; \Gamma \vdash \text{let}\rho \text{ in } R'[e]: \tau$

By assumption

2. $\Delta \uplus \{\rho\}; \Gamma \vdash R'[e]: \tau$

By (1) and inversion of htletr

3. $\Delta \uplus \{\rho\} \uplus \Delta''; \Gamma \vdash e: \tau$

By IH with (2)

Therefore $\Delta \uplus \Delta'; \Gamma \vdash e: \tau$

By (3) where $\Delta' = \{\rho\} \uplus \Delta''$

**Lemma 1.13 (Region Stack Replacement)** There exists $\Delta'$ such that if $\Delta \uplus \Delta'; \Gamma \vdash e_1: \tau$, $\Delta \uplus \Delta'; \Gamma \vdash e_2: \tau$, and $\Delta; \Gamma \vdash R[e_1]: \tau$ then $\Delta; \Gamma \vdash R[e_2]: \tau$.
Proof. By induction on the structure of $R$

**Induction Hypothesis:** There exists $\Delta'$ such that if $\Delta \uplus \Delta'; \Gamma \vdash e_1: \tau$, $\Delta \uplus \Delta'; \Gamma \vdash e_2: \tau$, and $\Delta; \Gamma \vdash R[e_1]: \tau$ then $\Delta; \Gamma \vdash R[e_2]: \tau$.

case $R = []$ When $\Delta' = \{\}$ because $R[e_2] = e_2$ and $\Delta \uplus \Delta' = \Delta$ therefore $\Delta; \Gamma \vdash e_2: \tau$ by assumption

case $R = \text{let} \; \rho \; \text{in} \; R'$ When $\Delta' = \{\rho\}$ because $R[e_1] = \text{let} \; \rho \; \text{in} \; R'[e_1]$

1. $\Delta \not\vdash \{\rho\}; \Gamma \vdash e_1: \tau$ By assumption
2. $\Delta \not\vdash \{\rho\}; \Gamma \vdash e_2: \tau$ By assumption
3. $\Delta; \Gamma \vdash \text{let} \; \rho \; \text{in} \; R'[e_1]: \tau$ By assumption
4. $\Delta \vdash \tau \text{ wf}$ By (3) and inversion of $\text{ht} \; \text{let} \; \rho$
5. $\Delta \not\vdash \{\rho\}; \Gamma \vdash R'[e_1]: \tau$ By (3) and inversion of $\text{ht} \; \text{let} \; \rho$
6. $\Delta \not\vdash \{\rho\} \not\vdash \Delta''; \Gamma \vdash e_1: \tau$ By Lemma 1.12 with (1)
7. $\Delta \not\vdash \{\rho\} \not\vdash \Delta''; \Gamma \vdash e_2: \tau$ By Lemma 1.12 with (2)
8. $\Delta \not\vdash \{\rho\}; \Gamma \vdash R'[e_2]: \tau$ By IH with (6), (7), and (3)

Therefore $\Delta; \Gamma \vdash \text{let} \; \rho \; \text{in} \; R'[e_2]: \tau$ By $\text{ht} \; \text{let} \; \rho$ with (4) and (8)
Syntax

\[
\begin{align*}
types \quad & \tau ::= \ldots \mid \text{bool}(b) \mid \text{bool} \\
terms \quad & e ::= \ldots \mid b \mid \text{isabool}(e) \mid (\text{if } e_1 \ e_2 \ e_3) \\
values \quad & v ::= \ldots \mid b \mid \text{isabool}(b) \\
booleans \quad & b ::= \text{true} \mid \text{false} \\
control contexts \quad & E ::= \ldots \mid \text{isabool}(E) \mid (\text{if } E \ e_1 \ e_2)
\end{align*}
\]

Expression Reductions

\[
\begin{align*}
\text{rdsiftrue} & \quad (\text{if \ isabool(true)} \ e_1 \ e_2) \rightarrow e_1 \\
\text{rdsiffalse} & \quad (\text{if \ isabool(false)} \ e_1 \ e_2) \rightarrow e_2
\end{align*}
\]

Figure 22: Booleans with subtyping

B Semantics for Forwading Pointers

In this section we formally describe our approach to forwarding pointers by exhibiting a “simple” language that provides safe covariant references. To make any formal safety claims we first must describe a language that has subtyping and region allocated mutable references. Doing this is best done in stages. We will first describe a system with a trivial subtyping relationship on boolean values, as an extension to our original region calculus. Then we will extend our language to include region allocated mutable references. Finally, we will describe how to provide safe covariant references by the addition of one new operator. The language we describe here is quite small and impractical for use in a real system. However, the language highlights the key ideas need to extend the system to include more non-trivial features.

B.1 Subtyping on Booleans

Figure 22 describes the syntax and dynamic semantics needed to provide a very simple form of subtyping over boolean values. We add two new type constructors \text{bool} and \text{bool}(b) the intuitive subtyping relationship is

\[
\begin{tikzpicture}[<->,>=stealth]
\node (b) {\text{bool}};
\node (bt) {\text{bool(true)}} at (0,0);
\node (bf) {\text{bool(false)}} at (1,0);
\draw (b) -- (bt) node[midway, below] {$\leq$};
\draw (b) -- (bf) node[midway, below] {$\leq$};
\end{tikzpicture}
\]

Figure 23 describes the needed extensions to our typing relations. Rather than adding a subsumption rule to our typing relation we introduce a coercion term \text{isabool}(e) which promotes an expression of type \text{bool}(b) into an expression of type \text{bool}. Note that the values of type \text{bool} are of the form \text{isabool}(b) while a value of type \text{bool}(b) is of the form \(b\). The if expression is defined over values of type \text{bool}.
Figure 23: Typing Judgements for Booleans with subtyping
This coercive interpretation of subtyping can be extended to include all the standard subtyping relationships as done by [9, 10]. Relying on explicit coercions rather than subsumption gives us control over what contexts subtyping can be used. We will see later that covariant references are only safe in a particular context.

B.2 Region Allocated References

Figure 24 describes the extensions to our semantics for region allocated references. A \((\tau \text{ ref } \rho)\) describes a mutable reference to a value of type \(\tau\) allocated in region \(\rho\). Rather than introducing a new syntactic category for locations, we simply use variables bound by a new expression \(\text{letl}\). Our semantics does not support mutual recursion, but by using \(\text{letl}\) bound variables to encode locations we avoid the need to add a type for heaps to our typing relation, which allows us to reuse many of our previous results unchanged in proving soundness for this extended calculus. The expression \(\text{deref}\) dereferences location into a value. The expression \(\text{update}[\rho] \; e_1 := e_2; e_3\) evaluates \(e_1\) to a location allocated in region \(\rho\). It then updates location with to value of \(e_2\) and continues by evaluating \(e_3\). Since locations are first class values we add them to the set of values also.

In our previous semantics region stacks were simply a set of nested \(\text{letr}\) bindings. To model references we extend region stacks so that between \(\text{letr}\) bindings there exists a possibly empty sequence of nested \(\text{letl}\) bindings which encodes a mapping between locations and values. We redefine the meaning of \(R^\Delta\) appropriately. We also introduce the notation \(R^\rho,H,x,\tau,v,H'\) which represents a region stack containing a bound region, \(\rho\), which contains a heap, \(H\), that includes a location, \(x\), bound to some value, \(v\), of type \(\tau\) and some arbitrary subheap \(H'\). This definition will be useful when defining our dynamic semantics. The rule \(\text{rdsderef}\) dereferences locations. The rule \(\text{rdsletupd}\) updates and existing binding. The rule \(\text{rdsletl}\) lifts’ \(\text{letl}\) bindings up to the appropriate heap.

Figure 25 contains the straightforward typing rules for our new constructs. Notice that by using variables bound by \(\text{letl}\) to encode locations we avoid the need for a separate heap type, and can type locations using the standard \(\text{htvar}\) rule.

B.3 Covariant References

Now that we have a calculus with a very basic form of subtyping and mutable references, we can introduce a new term to our language that allows for safe covariant references. Figure 26 describes the needed extensions. The term \(\text{gclemma}\) evaluates an expression \(e_1\) to a value of type \((\text{bool}(b) \text{ ref } \rho_1)\) it then coerces that value into a value of type \((\text{bool ref } \rho_2)\) where \(\rho_2\) is a new freshly bound region variable. It then binds this value to the some variable, \(x\), and evaluates the body, \(e_2\), in a restricted region environment \(\Delta\). The reduction rule \(\text{rdslemma}\) is safe if and only if \(x'\) does not occur in the body of \(e\). We can be assured of this fact when \(\rho_1\) is not in \(\Delta\). If \(x'\) did occur in \(e\) and \(\rho_1\) is not in \(\Delta\) this would violate the assumption that \(e\) is well typed. Figure 27 describes the typing rule needed as a precondition for safety. Notice that we statically require that \(\rho_1\) statically not be a member of \(\Delta_2 \cup \{\rho_2\}\), however this is a necessary but not sufficient restriction to guarantee safety, because of region aliasing. We must check at runtime that aliasing has not violated this constraint. If it has we must abort
Syntax

\[
\begin{align*}
types \quad \tau & ::= \ldots \mid (\tau \text{ ref } \rho) \quad \text{ region annotated reference} \\
terms \quad e & ::= \ldots \\
\quad & \mid \text{letl } x: (\tau \text{ ref } \rho) = e_1 \text{ in } e_2 \quad \text{ bind new location} \\
\quad & \mid \text{deref}[\rho](e) \quad \text{ dereference location} \\
\quad & \mid \text{update}[\rho] e_1 := e_2; e_3 \quad \text{ update location and continue} \\
values \quad v & ::= \ldots \mid x \\
control contexts \quad E & ::= \ldots \\
\quad & \mid \text{letl } x: (\tau \text{ ref } \rho) = E \text{ in } e \\
\quad & \mid \text{deref}[\rho](E) \\
\quad & \mid \text{update}[\rho] E := e_1; e_2 \mid \text{update}[\rho] v := E; e \\
region stacks \quad R & ::= [\square] \mid \text{letr } \rho \text{ in } H \\
heaps \quad H & ::= R \mid \text{letl } x: (\tau \text{ ref } \rho) = v \text{ in } H \\
\end{align*}
\]

Program Reductions

\[
\begin{align*}
[\square]^\Delta & \deft [\square] \\
(\text{letr } \rho \text{ in } H)^{\Delta w(\rho)} & \deft (\text{letr } \rho \text{ in } H^{\Delta w(\rho)}) \\
(\text{letl } x: (\tau \text{ ref } \rho) = H \text{ in } )^{\Delta w(\rho)} & \deft (\text{letl } x: (\rho \text{ ref } \tau) = H^{\Delta w(\rho)} \text{ in } ) \\
(\text{letl } x: (\tau \text{ ref } \rho) = H \text{ in } )^{\Delta} & \deft H^{\Delta} \quad \text{where } \rho \notin \Delta \\
\end{align*}
\]

\[
R^{\rho,H,x,\tau,v,H'} \deft R[\text{letr } \rho \text{ in } H[\text{letl } x: (\tau \text{ ref } \rho) = v \text{ in } H']] \\
\]

\[
\begin{align*}
\text{rdsderef} & \quad R^{\rho,H,x,\tau,v,H'}[E[\text{deref}[\rho](x)]] \rightarrow_p R^{\rho,H,x,\tau,v,H'}[E[x]] \\
\text{rdsletupd} & \quad R^{\rho,H,x,\tau,v,H'}[E[\text{update}[\rho](x) := v'; e]] \rightarrow_p R^{\rho,H,x,\tau,v,H'}[E[e]] \\
\text{rdsletl} & \quad R[\text{letr } \rho \text{ in } H[R'[E[\text{letl } x: (\rho \text{ ref } \tau) = v \text{ in } e]]]] \rightarrow_p R^{\rho,H,x,\tau,v,H'}[E[e]] \\
\end{align*}
\]

Figure 24: Regions and References
\( \Delta; \Gamma \vdash e : \tau \)

\[
\begin{align*}
\Delta \vdash (\tau_1 \text{ref} \ \rho) \ \text{wf} & \quad \Delta; \Gamma \cup \{x : (\tau_1 \text{ref} \ \rho)\} \vdash e_1 : \tau_1 & \quad \Delta; \Gamma \cup \{x : (\tau_1 \text{ref} \ \rho)\} \vdash e_2 : \tau_2 \\
\Delta; \Gamma \vdash (\text{letl} \ x : (\tau_1 \text{ref} \ \rho) = e_1 \ \text{in} \ e_2) : \tau_2 & \quad \Delta \cup \{\rho\}; \Gamma \vdash e : (\tau \text{ref} \ \rho) & \quad \Delta \cup \{\rho\}; \Gamma \vdash \text{deref}[\rho] (e) : \tau \\
\Delta \cup \{\rho\}; \Gamma \vdash e_1 : (\tau_1 \text{ref} \ \rho) & \quad \Delta \cup \{\rho\}; \Gamma \vdash e_2 : \tau_1 & \quad \Delta \cup \{\rho\}; \Gamma \vdash e_3 : \tau_2 \\
\Delta \cup \{\rho\}; \Gamma \vdash (\text{update}[\rho] \ e_1 := e_2; e_3) : \tau_2 & \quad \Delta \vdash \tau \ \text{wf} \\
\end{align*}
\]

\[
\begin{align*}
\Delta \vdash \tau \ \text{wf} & \quad \cdots \\
\Delta \cup \{\rho\} \vdash \tau \ \text{wf} & \quad \Delta \cup \{\rho\} \vdash (\tau \text{ref} \ \rho) \ \text{wf} \\
\end{align*}
\]

Figure 25: Typing Judgements for Regions and References
Syntax

\[
\begin{align*}
terms \quad e & ::= \ldots \\
  & \mid \text{gclemma} \\
  & \quad \text{assume (bool}(b) \text{ ref } \rho_1) \text{ isa (bool ref } \rho_2) \text{ in} \\
  & \quad \text{let } x = e_1 \text{ in} \\
  & \quad \text{only } \Delta \text{ in } e_2 \\
control contexts \quad E & ::= \ldots \\
  & \mid \text{gclemma} \\
  & \quad \text{assume (bool}(b) \text{ ref } \rho_1) \text{ isa (bool ref } \rho_2) \text{ in} \\
  & \quad \text{let } x = E \text{ in} \\
  & \quad \text{only } \Delta \text{ in } e \\
\end{align*}
\]

\[
\text{rdsgclemma } R^{\rho_1,H,x',\text{bool}(b),h'}[E]\text{gclemma} \\
  \text{assume (bool}(b) \text{ ref } \rho_1) \text{ isa (bool ref } \rho_2) \text{ in} \\
  \text{let } x = x' \text{ in} \\
  \text{only } \Delta \text{ in } e ]] \mapsto_p R'[E[\text{only } \Delta \text{ in } e[x'/x]]] \\
\text{where } R' = R^{\rho_1,H,x',\text{bool},\text{isabool}(b),h'}[\rho_2/\rho_1] \text{ and } \rho_1 \not\in \Delta \\
\text{rdsglemmaerr } R[E]\text{gclemma} \\
  \text{assume (bool}(b) \text{ ref } \rho_1) \text{ isa (bool ref } \rho_2) \text{ in} \\
  \text{let } x = v \text{ in} \\
  \text{only } \Delta \uplus \{ \rho_1 \} \text{ in } e ]] \mapsto_p \text{halt}^{\text{Ans}}
\]

Figure 26: Covariant References

\[
\Delta; \Gamma \vdash e: \tau
\]

\[
\Delta_1 \uplus \{ \rho_1 \} \vdash \Gamma_1 \text{ wenv} \\
\Delta_1 \uplus \{ \rho_1 \} \uplus \Delta_2; \Gamma_1 \uplus \Gamma_2 \vdash e_1 : (\text{bool}(b) \text{ ref } \rho_1) \\
\Delta_2 \uplus \{ \rho_2 \}; \Gamma_2 \uplus \{ x : (\text{bool ref } \rho_2) \} \vdash e_2 : \text{Ans} \\
\hline \\
\Delta_1 \uplus \{ \rho_1 \} \uplus \Delta_2; \Gamma_1 \uplus \Gamma_2 \vdash (\text{gclemma} \\
  \text{assume (bool}(b) \text{ ref } \rho_1) \text{ isa (bool ref } \rho_2) \text{ in} \\
  \text{let } x = e_1 \text{ in} \\
  \text{only } \Delta_2 \uplus \{ \rho_2 \} \text{ in } e_2 : \text{Ans}
\]

Figure 27: Typing Judgements for Covariant References

54
our computation using the \texttt{rdsglemmaerr} rule.