Type-Preserving Garbage Collectors (Extended Version)

Daniel C. Wang Andrew W. Appel

Department of Computer Science
Princeton University
Princeton, NJ 08544 USA

Tech Report TR-624-00

Abstract

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

Since our language is strongly typed, the standard type soundness guarantee "Well typed programs do not go wrong" is extended to include the collector. Our type safety guarantee is non-trivial since not only does it guarantee the type safety of the garbage collector, but it guarantees that the collector preservers the type safety of the program being garbage collected. We describe the technique in detail and report performance measurements for a few microbenchmarks. 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

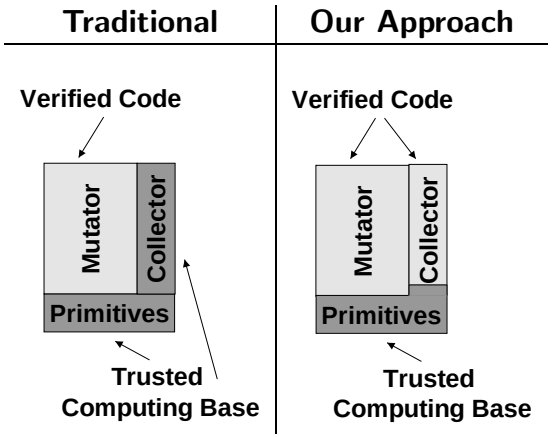

Figure 1: Reduced Trusted Computing Base

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)
```

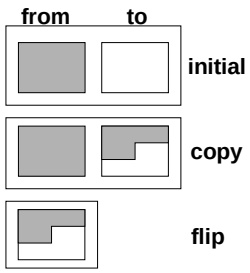

Figure 2: Traditional Garbage Collector

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

In Section 2 we 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 and 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*[2] 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.

[2]Early in the sense of sooner than what a strict LIFO policy would allow.

$$
\begin{array}{rrcll}
\textit{types} & \tau & ::= & \mathsf{unit} & \\
 & & | & \tau_1 \xrightarrow{\Delta} \tau_2 & \text{function type with effect} \\
 & & | & \forall\rho.\tau & \text{region polymorphic type} \\
 & & | & (\tau\ \mathsf{at}\ \rho) & \text{region annotated type} \\
 & & | & \mathsf{Ans} & \\
\textit{terms} & e & ::= & x & \\
 & & | & \langle\rangle & \\
 & & | & (\lambda x{:}\tau.e)^{\Delta} & \text{effect annotated function} \\
 & & | & (e_1\ e_2) & \\
 & & | & (\Lambda\rho.e) & \text{region abstraction} \\
 & & | & (e[\rho]) & \text{region application} \\
 & & | & \mathsf{letr}\ \rho\ \mathsf{in}\ e & \text{allocate new region} \\
 & & | & \mathsf{put}[\rho](e) & \text{put value in region} \\
 & & | & \mathsf{get}[\rho](e) & \text{get value from region} \\
 & & | & \mathsf{only}\ \Delta\ \mathsf{in}\ e & \text{deallocate regions early} \\
 & & | & (\mathsf{fix} f{:}\tau.v) & \\
 & & | & \mathsf{halt}^{\tau} & \\
\textit{values} & v & ::= & \langle\rangle \mid (\lambda x{:}\tau.e)^{\Delta} \mid (\Lambda\rho.e) \mid \mathsf{put}[\rho](v) & \\
\textit{region contexts} & \Delta & ::= & \{\} \mid \{\rho_1, \ldots, \rho_n\} &
\end{array}
$$

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 $\mathsf{put}[\rho](e)$, which evaluates it argument e and stores the resulting value into region ρ. If the type of the argument e is of type τ the type of the resulting value is $(\tau\ \mathsf{at}\ \rho)$. The type constructor at describes objects of some type τ allocated in region ρ. The operator $\mathsf{get}[\rho](e')$ expects that it's argument e' evaluates to a value of type $(\tau\ \mathsf{at}\ \rho)$. It then fetches the value from region ρ and returns a value of type τ. The terms $\mathsf{put}[\rho](e)$ and $\mathsf{get}[\rho](e)$ 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 ρ is represented by the syntactic term $\mathsf{put}[\rho](v)$. We treat region contexts Δ as sets of region variables.

When the body of function $(\lambda x{:}\tau.e)^{\Delta}$ captures values in its closure we 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 Δ 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 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 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 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 halt^τ 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 rdspure. The global program reduction rule rdsletr hoists an inner letr binding to the top-level region stack. Since we assume all bound variables are unique, the region in the letr binding is guaranteed to be a fresh variable. The rdsget rule converts a value allocated in region ρ to a pure value, if the region ρ is currently bound by the enclosing region stack. The rdsonly rule throws away the surrounding control context and continues evaluating its body in a new region stack defined by the only expression. The notation R^Δ is the smallest region stack binding the variables in Δ.

The rdshalt rule immediately throws away any non-trivial surrounding control context and region stack and reduces to an answer. The rdsfree 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 rdsfree implicitly allows for non-LIFO allocation policy, already providing support for early deallocation, which make make the rdsonly rule seem redundant. However, if we did not include the 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.

$$
\begin{array}{rrcl}
\textit{programs} & P & ::= & R[e] \\
\textit{answers} & A & ::= & R[v] \mid \mathsf{halt}^{\tau} \\
\textit{control contexts} & E & ::= & [\,] \\
 & & \mid & (E\ e) \mid (v\ E) \\
 & & \mid & (E[\rho]) \\
 & & \mid & \mathsf{put}[\rho](E) \mid \mathsf{get}[\rho](E) \\
\textit{region stacks} & R & ::= & [\,] \mid \mathsf{letr}\ \rho\ \mathsf{in}\ R
\end{array}
$$

Expression Reductions

$$
\begin{array}{rrcl}
\mathsf{rdsbetav} & ((\lambda x{:}\tau.e)^{\Delta}\ v) & \mapsto_e & e[v/x] \\
\mathsf{rdstapp} & ((\Lambda\rho.e)[\rho']) & \mapsto_e & e[\rho'/\rho] \\
\mathsf{rdsfix} & (\mathsf{fix} f{:}\tau.v) & \mapsto_e & v[(\mathsf{fix} f{:}\tau.v)/f]
\end{array}
$$

Program Reductions

$$
\begin{array}{rrcl}
\mathsf{rdspure} & R[E[e_1]] & \mapsto_P & R[E[e_2]] \ \text{where}\ e_1 \mapsto_e e_2 \\
\mathsf{rdsletr} & R[E[\mathsf{letr}\ \rho\ \mathsf{in}\ e]] & \mapsto_P & R[\mathsf{letr}\ \rho\ \mathsf{in}\ E[e]] \\
\mathsf{rdsget} & R[\mathsf{letr}\ \rho\ \mathsf{in}\ R'[E[\mathsf{get}[\rho](\mathsf{put}[\rho](v))]]] & \mapsto_P & R[\mathsf{letr}\ \rho\ \mathsf{in}\ R'[E[v]]] \\
\mathsf{rdsonly} & R[E[\mathsf{only}\ \Delta\ \mathsf{in}\ e]] & \mapsto_P & R'^{\Delta}[e] \\
\mathsf{rdshalt} & R[E[\mathsf{halt}^{\tau'}]] & \mapsto_P & \mathsf{halt}^{\tau} \ \text{where}\ E \neq [\,] \\
\mathsf{rdsfreer} & R[\mathsf{letr}\ \rho\ \mathsf{in}\ R'[e]] & \mapsto_P & R[R'[e]] \ \text{where}\ \vdash R[R'[e]]\ \mathrm{wt}
\end{array}
$$

$$
\begin{array}{rcl}
[\,]^{\Delta} & \stackrel{def}{=} & [\,] \\
(\mathsf{letr}\ \rho\ \mathsf{in}\ R)^{\Delta \uplus \{\rho\}} & \stackrel{def}{=} & (\mathsf{letr}\ \rho\ \mathsf{in}\ R^{\Delta \uplus \{\rho\}}) \\
(\mathsf{letr}\ \rho\ \mathsf{in}\ R)^{\Delta} & \stackrel{def}{=} & R^{\Delta} \ \text{where}\ \rho \notin \Delta
\end{array}
$$

Multi-step Reduction

$$
\frac{}{P \mapsto_P^* P}\ \mathsf{mstprefl} \qquad \frac{P_1 \mapsto_P^* P_2 \quad P_2 \mapsto_P^* P_3}{P_1 \mapsto_P^* P_3}\ \mathsf{mstptrans} \qquad \frac{P_1 \mapsto_P P_2}{P_1 \mapsto_P^* P_2}\ \mathsf{mstprds}
$$

Figure 4: Dynamic Semantics

$$\textit{value environment} \quad \Gamma \quad ::= \quad \{\} \mid \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$$

Judgement	Meaning
$\vdash P$ wt	P is a well typed program.
$\Delta; \Gamma \vdash e : \tau$	e has type τ under $\Delta; \Gamma$. We always implicitly require $\Delta \vdash \Gamma$ wfenv
$\Delta \vdash \tau$ wf	τ is a well formed type with respect to Δ.
$\Delta \vdash \Gamma$ wfenv	Γ is a well formed value environment with respect to Δ.

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 τ where x does not occur already bound in Γ.

The judgment $\vdash P$ 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 τ under the value environment Γ and the region context Δ. The judgment $\Delta \vdash \tau$ wf asserts that all the free region variables in τ occur in Δ. The judgment $\Delta \vdash \Gamma$ 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 Γ occur in the region context Δ. That is we omit the explicit constraint that $\Delta \vdash \Gamma$ wfenv for many of the rules. For the non-trivial rules such htonly and htabs we make the $\Delta \vdash \Gamma$ 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 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 htonly rule. A simple inductive argument will establish the fact that any closed program of type Ans which evaluates to an answer must evaluate to the answer $\mathsf{halt}^{\mathsf{Ans}}$. So informally speaking the requirement that the body of the only Δ in e expression be of type 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 htonly and htabs check subexpressions in a non-standard way the standard substitution lemma for terms does not hold. A more restrictive substitution for values and fix expressions does hold. Since our calculus is call-by-value the more restrictive

$\boxed{\vdash P \text{ wt}}$

$$\frac{\{\};\{\} \vdash P : \tau}{\vdash P \text{ wt}}\ \mathsf{wte}$$

$\boxed{\Delta;\Gamma \vdash e : \tau}$

$$\frac{}{\Delta;\Gamma \uplus \{x:\tau\} \vdash x:\tau}\ \mathsf{htvar} \qquad \frac{}{\Delta;\Gamma \vdash \langle\rangle : \mathsf{unit}}\ \mathsf{htunit}$$

$$\frac{\Delta \vdash \Gamma \text{ wfenv} \quad \Delta' \vdash \tau_1 \text{ wf} \quad \Delta';\Gamma' \uplus \{x:\tau_1\} \vdash e:\tau_2}{\Delta \uplus \Delta';\Gamma \uplus \Gamma' \vdash (\lambda x:\tau_1.e)^{\Delta'} : \tau_1 \xrightarrow{\Delta'} \tau_2}\ \mathsf{htabs}$$

$$\frac{\Delta \uplus \Delta';\Gamma \vdash e_1 : \tau_1 \xrightarrow{\Delta'} \tau_2 \quad \Delta \uplus \Delta';\Gamma \vdash e_2 : \tau_1}{\Delta \uplus \Delta';\Gamma \vdash (e_1\ e_2) : \tau_2}\ \mathsf{htapp}$$

$$\frac{\Delta \uplus \{\rho\};\Gamma \vdash e:\tau}{\Delta;\Gamma \vdash (\Lambda\rho.e) : \forall\rho.\tau}\ \mathsf{httabs} \qquad \frac{\Delta \uplus \{\rho'\};\Gamma \vdash e : \forall\rho.\tau}{\Delta \uplus \{\rho'\};\Gamma \vdash (e[\rho']) : \tau[\rho'/\rho]}\ \mathsf{httapp}$$

$$\frac{\Delta \vdash \tau \text{ wf} \quad \Delta \uplus \{\rho\};\Gamma \vdash e:\tau}{\Delta;\Gamma \vdash (\mathsf{letr}\ \rho\ \mathsf{in}\ e) : \tau}\ \mathsf{htletr}$$

$$\frac{\Delta \uplus \{\rho\};\Gamma \vdash e:\tau}{\Delta \uplus \{\rho\};\Gamma \vdash \mathsf{put}[\rho](e) : (\tau\ \mathsf{at}\ \rho)}\ \mathsf{htput} \qquad \frac{\Delta \uplus \{\rho\};\Gamma \vdash e : (\tau\ \mathsf{at}\ \rho)}{\Delta \uplus \{\rho\};\Gamma \vdash \mathsf{get}[\rho](e) : \tau}\ \mathsf{htget}$$

$$\frac{\Delta \vdash \Gamma \text{ wfenv} \qquad \Delta';\Gamma' \vdash e : \mathsf{Ans}}{\Delta \uplus \Delta';\Gamma \uplus \Gamma' \vdash (\mathsf{only}\ \Delta'\ \mathsf{in}\ e) : \mathsf{Ans}}\ \mathsf{htonly}$$

$$\frac{\Delta \vdash \tau \text{ wf} \quad \Delta;\Gamma \uplus \{f:\tau\} \vdash v:\tau}{\Delta;\Gamma \vdash (\mathsf{fix} f:\tau.v) : \tau}\ \mathsf{htfix} \qquad \frac{\Delta \vdash \tau \text{ wf}}{\Delta;\Gamma \vdash \mathsf{halt}^{\tau} : \tau}\ \mathsf{hthalt}$$

Figure 6: Main Typing Judgement

$\boxed{\Delta \vdash \tau \text{ wf}}$

$$\frac{}{\Delta \vdash \mathsf{unit} \text{ wf}}\ \mathsf{wfunit}$$

$$\frac{\Delta \vdash \tau_1 \text{ wf} \quad \Delta \vdash \tau_2 \text{ wf}}{\Delta \vdash \tau_1 \xrightarrow{\Delta} \tau_2 \text{ wf}}\ \mathsf{wfarrow} \qquad \frac{\Delta \uplus \{\rho\} \vdash \tau \text{ wf}}{\Delta \vdash \forall \rho.\tau \text{ wf}}\ \mathsf{wfall}$$

$$\frac{\Delta \uplus \{\rho\} \vdash \tau \text{ wf}}{\Delta \uplus \{\rho\} \vdash (\tau \ \mathsf{at}\ \rho) \text{ wf}}\ \mathsf{wfat}$$

$$\frac{}{\Delta \vdash \mathsf{Ans} \text{ wf}}\ \mathsf{wfAns}$$

$\boxed{\Delta \vdash \Gamma \text{ wfenv}}$

$$\frac{}{\Delta \vdash \{\} \text{ wfenv}}\ \mathsf{wfenvempty} \qquad \frac{\Delta \vdash \Gamma \text{ wfenv} \quad \Delta \vdash \tau \text{ wf}}{\Delta \vdash \Gamma \uplus \{x\!:\!\tau\} \text{ wfenv}}\ \mathsf{wfenvbv}$$

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 adopt a "region passing" semantics, where region variables are bound to dynamic values at runtime, and therefore region abstractions can not be erased. The only expression takes an arbitrary set of region variables. At runtime we simply note what regions are dynamically bound to the region variables passed to the only expression and safely deallocate any other regions, since they are not needed to evaluate the rest of the program. 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 system 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

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

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

The expression put$[\rho](1)$ stores the integer into the region ρ_1 and returns a reference to the integer. Notice that if we executes the first branch of the conditional then f behaves as expected. However, if we execute the second branch then at runtime the region variables ρ_a and ρ_b are both bound to the same region and the program will attempt to access a region which we have erroneously deallocated. To handle this situation correctly we can simply prevent programs 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

```
fun f[ρa, ρb](x:int at ρb):Ans =
  only ρb in (get[ρb](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 reserver 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 counting approach that requires updating a reference count for each interregion store. Because our type system provides more guarantees we can safely deallocate regions without needing to maintain any reference counts.

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 $P_1 \mapsto_P P_2$.*

The main safety theorem is as follows

Theorem 1 (Type Soundness) *If $\vdash P_1$ wt then, there is no stuck P_2 such that $P_1 \mapsto_P^* P_2$.*

The proof of our main theorem follows directly with a slightly stronger induction hypothesis over the $\mapsto_P^*$ relation and the application of Lemma 1.1 and Lemma 1.2.

Lemma 1.1 (Type Preservation of Programs) *If $\vdash P_1$ wt and $P_1 \mapsto_P P_2$, then $\vdash P_2$ wt.*

Lemma 1.2 (Progress) *If $\vdash P_1$ wt then, there exists P_2 such that $P_1 \mapsto_P P_2$ 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 $\Delta; \Gamma \uplus \{x : \tau\} \vdash e' : \tau'$ then $\Delta; \Gamma \vdash e'[e/x] : \tau'$, where $e = v$ or $e = (\mathsf{fix} f : \tau''.v)$.*

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 $\Delta' \vdash \tau$ wf, $\Delta \vdash \Gamma$ wfenv, $\Delta' \vdash \Gamma'$ wfenv, and $\Delta \uplus \Delta'; \Gamma \uplus \Gamma' \vdash e : \tau$ where $e = v$ or $e = (\mathsf{fix} f : \tau.v)$ 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* $\Delta;\{\} \vdash e_1 : \tau$ *and* $e_1 \mapsto_e e_2$*, then* $\Delta;\{\} \vdash e_2 : \tau$*.*

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

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

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

Lemma 1.8 (Typing Relation Preservers Well Formedness) *If* $\Delta;\Gamma \vdash e : \tau$ *then* $\Delta \vdash \tau$ 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* Δ' *such that* $\Delta \uplus \Delta';\Gamma \vdash e:\tau$*.*

Lemma 1.13 (Region Stack Replacement) *There exists* Δ' *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
```

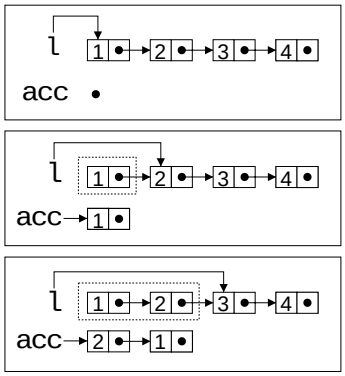

Figure 8: Iterative List Reverse

2.6 An Extended First-Order Variant

We can easily extend our core calculus to a 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 cannont copy values captured by the closure of a higer-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 when the program first calls `itrev` the call is not a tail call, so we must allocate a trivial stack frame for this call. As the function recursively descends `l` the previous list cells, contained in the dotted box in the figure, are garbage 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

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

Figure 9: First-order Closure Conversion

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_rl` which is needed for our one non-tail call in the original program. In general each call site of `itrev` will require one new data constructor to represent each distinct return continuation. 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

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

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

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

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

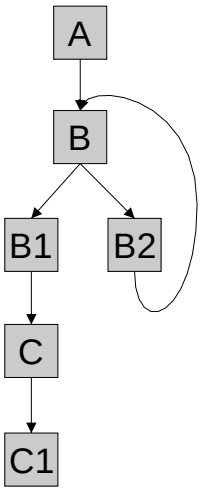

Figure 10: CPS-Converted and Closure-Converted Program

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 `lst` in figure 10 becomes a type constructor `lst[`ρ`]` parameterized by a region in which the list lives. Since we can represent both the empty list and return continuation as single machine words we do not need to allocate space for them. We need to allocate space only when constructing list cells with the `Cons` data-constructor; this is reflected in the type `Cons(int, lst[`ρ`]) at` ρ.

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

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

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

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

Figure 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
type cont[ρ] = Ret_rl
type gc_cont[ρ] = Ret_itrev(cont[ρ], lst[ρ], lst[ρ]) at ρ

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

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

Figure 13: "Flipping" from and to space

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

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

Figure 14: Copying roots

the region dynamically bound to ρ_{to}.

In this case it is obvious that ρ_{to} is distinct from ρ_{from} as it is a new fresh region. A static type system that tracks the uniqueness of regions such as [11] would be sufficient, and our dynamic apporach is not strictly necessary in this case. However, since the extra runtime overhead is negligible, we prefer our simpler dynamic approach to the more complicated static system. We believe that we can integrate the explicit deallocation techniques that use static typing to prevent region aliasing with our implicit approach to give us the benefits of both approaches, so that we resort to this dynamic approach when we are unable statically determine aliasing relationships.

GC copy function. Figure 14 sketches the code for a naive copy function. The type of the copy function guarantees that the function performs a deep copy. The copy function is not written in continuation-passing style so it uses a stack while traversing the list. We could write the copy function in continuation-passing style and heap-allocate all its temporary space in a third region which we could reclaim after we are done. Alternatively if we extend our type system with enough technical machinery so that we can recycle the space used by the continuations we could implement what would amount to the Deutsch-Schorr-Waite pointer reversal algorithm [24, 30, 26, 32]. Note that the function `copy_cont` performs an operation equivalent to "walking the stack". Since we have CPS converted our program the continuation, `k`, represents the current stack frame. It may be the case that we can adapt the higher-order techniques to provide true abstraction and separate compilation in the presence of a garbage collector by requiring each abstract object to provide a method[4] to copy or trace the object. It is not clear what the software engineering and performance issues are for this technique so we consider it to be future work. A more serious problem with our copy function is that it does not preserve pointer sharing.

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

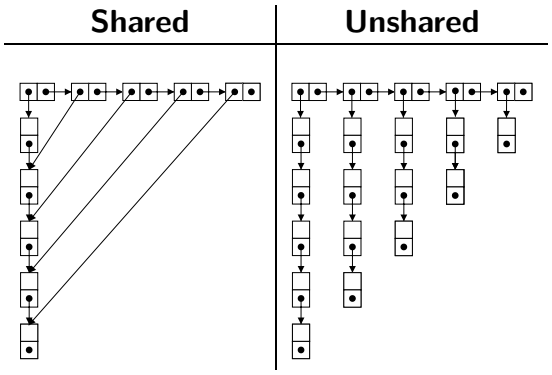

Figure 15: Shared vs. Unshared values

Preserving Sharing. Consider the datastructure 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 `gc_lst` type describes the garbage collector's view of lists. From the garbage collector's standpoint, lists are allocated in a from-space containing forwarding pointers into objects in a to-space. It must be the case that that lists allocated in the to-space have forwarding pointers which are always set to `NULL`. The `lst` type describes lists that the mutator operates on, and maintains the invariant that the forwarding pointer is set to `NULL`. The fact that the forwarding pointer is a mutable field which the garbage collector will mutate is captured by the use of the ref constructor.

```
prim objId : [α] 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] =
 case x of Nil ⇒ lst[ρto].Nil
 | Cons(hd, tl) ⇒
    if inTbl[ρtbl, lst[ρfrom], lst[ρto]](x) then (* is forwarded? *)
     getTbl[ρtbl, lst[ρfrom], lst[ρto]](x)
     else let hd' = hd in
      let tl' = share_copy_lst[ρtbl, ρfrom, ρto](t,tl) in
      let x' = lst[ρto].Cons(hd',tl')
      in addTbl[ρtbl, lst[ρfrom], lst[ρto]](x,x') ; (* set forwaded *)
         x'
...
```

Figure 16: Preserving Sharing with a Hash-Table

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

Figure 17: Encoding Forwading 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[5] 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

[5]A 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 $\delta - main$ 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

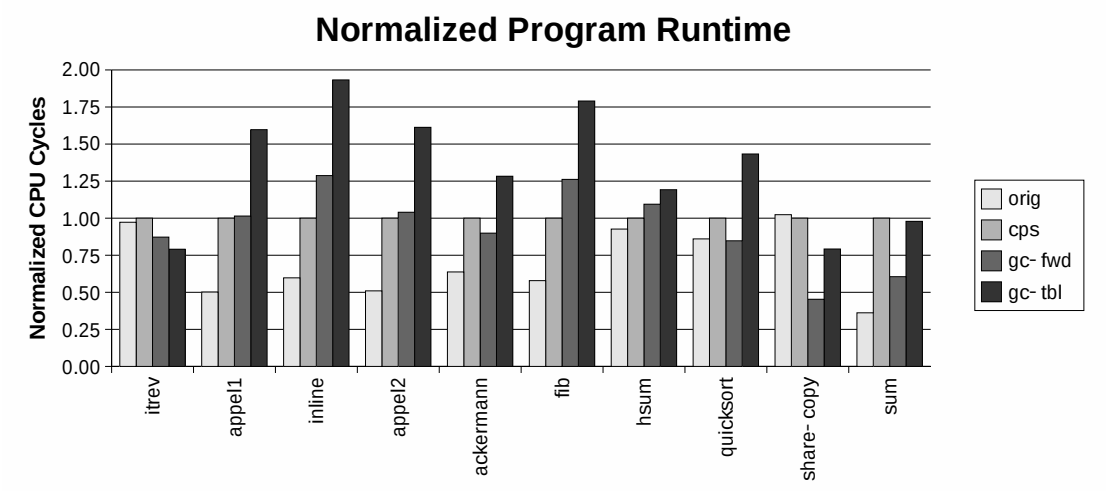

Figure 18: Relative Runtime Performance

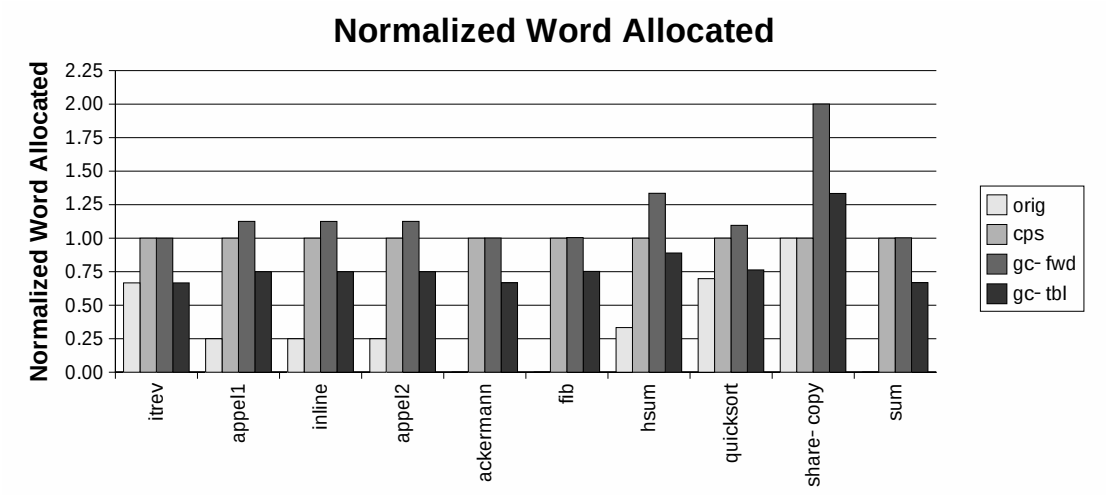

Figure 19: Relative Number of Words Allocated

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

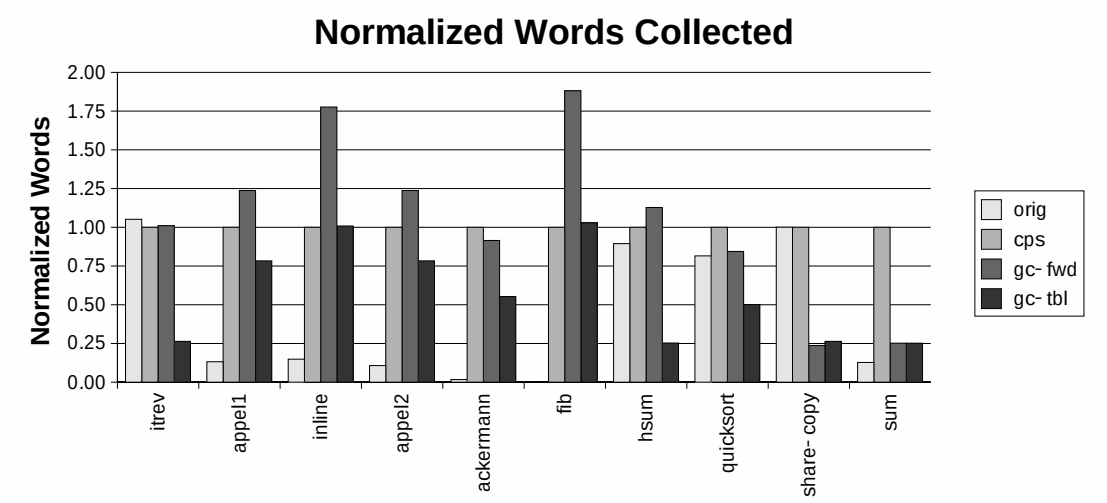

Figure 20: Relative Number of Words Garbage Collected

shows that our CPS converted programs are allocating significantly more heap data[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.[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.[8] In this case

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

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

[8]In the case of `share-copy`, which is allocating more data, because of our dynamic heap resizing policy

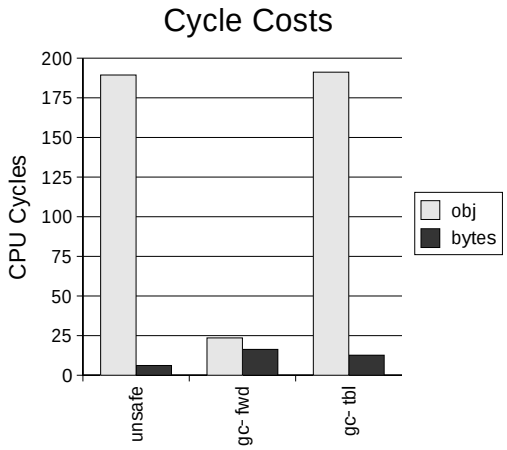

Figure 21: Per-word and per-object cycle costs

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

We would like to investigate how to integrate purely static memory management techniques [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 management: Improving region-based analysis of higher-order languages. In *Proceedings of SIGPLAN'95 Conference on Programming Languages Design and Implementation*, volume 30 of *ACM SIGPLAN Notices*, pages 174–185, La Jolla, CA, June 1995. ACM Press.

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

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

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

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

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

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

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

[9] Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae*, 33:309–338, 1998.

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

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

[12] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In ICFP [17], pages 301–312.

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

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

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

[16] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of the Twenty-second Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 130–141. ACM Press, January 1995.

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

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

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

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

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

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

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

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

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

[26] Jonathan Sobel and Daniel P. Friedman. Recycling continuations. In ICFP [17], pages 251–270.

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

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

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

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

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

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

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

[34] Wright and Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

A Proof of Safety Properties

Theorem 1 (Type Soundness) *If* $\vdash P_1$ 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* $\vdash P_1$ wt *and* $P_1 \mapsto_P^* P_2$ *then* P_2 *is not stuck and* $\vdash P_2$ wt.

case $\boxed{\dfrac{}{P \mapsto_P^* P}\ \mathsf{mstprefl}}$ By assumption $\vdash P$ wt and by Lemma 1.2 P is not stuck.

case $\boxed{\dfrac{P_1 \mapsto_P^* P_2 \quad P_2 \mapsto_P^* P_3}{P_1 \mapsto_P^* P_3}\ \mathsf{mstptrans}}$

1. $\vdash P_1$ wt — By assumption
2. $P_1 \mapsto_P^* P_2$ — By assumption
3. $P_2 \mapsto_P^* P_3$ — By assumption
4. $\vdash P_2$ wt — By IH with (1) and (2)

   It follows that $\vdash P_3$ wt and P_3 is not stuck — By IH with (4) and (3)

case $\boxed{\dfrac{P_1 \mapsto_P P_2}{P_1 \mapsto_P^* P_2}\ \mathsf{mstprds}}$

1. $\vdash P_1$ wt — By assumption
2. $P_1 \mapsto_P P_2$ — By assumption
3. $\vdash P_2$ wt — By Lemma 1.1 with (1) and (2)
4. P_2 is not stuck — By Lemma 1.2 and (3)

   It follows that $\vdash P_2$ wt and P_2 is not stuck — By (3) and (4)

Lemma 1.1 (Type Preservation of Programs) *If* $\vdash P_1$ wt *and* $P_1 \mapsto_P P_2$, *then* $\vdash P_2$ wt.

Proof. By case analysis of $P_1 \mapsto_P P_2$

Consdier the cases of $P_1 \mapsto_P P_2$

case $\boxed{\mathsf{rdspure}\ \ R[E[e_1]] \mapsto_P R[E[e_2]] \text{ where } e_1 \mapsto_e e_2}$

1. $\vdash R[E[e_1]]$ wt — By assumption
2. $e_1 \mapsto_e e_2$ — By assumption
3. $\{\};\{\} \vdash R[E[e_1]] : \tau$ — By (1) and inversion of wte

4. $\{\} \uplus \Delta'; \{\} \vdash E[e_1] : \tau$ — By Lemma 1.12 and (3)
5. $\{\} \uplus \Delta'; \{\} \vdash e_1 : \tau'$ — By Lemma 1.10 and (5)
6. $\{\} \uplus \Delta'; \{\} \vdash e_2 : \tau'$ — By Lemma 1.5 with (5) and (2)
7. $\{\} \uplus \Delta'; \{\} \vdash E[e_2] : \tau$ — By Lemma 1.11 with (5), (6), and (4)
8. $\{\}; \{\} \vdash R[E[e_2]] : \tau$ — By Lemma 1.13 with (4), (7), and (3)

Therefore $\vdash R[E[e_2]]$ wt — By wte and (8)

case rdsletr $R[E[\mathsf{letr}\ \rho\ \mathsf{in}\ e]] \mapsto_P R[\mathsf{letr}\ \rho\ \mathsf{in}\ E[e]]$

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

Therefore $\vdash R[\mathsf{letr}\ \rho\ \mathsf{in}\ E[e]]$ wt — By wte and (11)

case rdsget $R[\mathsf{letr}\ \rho\ \mathsf{in}\ R'[E[\mathsf{get}[\rho](\mathsf{put}[\rho](v))]]] \mapsto_P R[\mathsf{letr}\ \rho\ \mathsf{in}\ R'[E[v]]]$

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

Therefore $\vdash R[\mathsf{letr}\ \rho\ \mathsf{in}\ R'[E[v]]]$ wt — By wte and (8)

case rdsonly $R[E[\mathsf{only}\ \Delta''\ \mathsf{in}\ e]] \mapsto_P R'^{\Delta''}[e]$

1. $\vdash R[E[\mathsf{only}\ \Delta''\ \mathsf{in}\ e]]$ wt — By assumption

2. $\{\};\{\} \vdash R[E[\mathsf{only}\ \Delta''\ \mathsf{in}\ e]] : \tau$ — By (1) and inversion of wte
3. $\{\} \uplus \Delta;\{\} \vdash E[\mathsf{only}\ \Delta''\ \mathsf{in}\ e] : \tau$ — By Lemma 1.12 with (2)
4. $\{\} \uplus \Delta;\{\} \vdash \mathsf{only}\ \Delta''\ \mathsf{in}\ e : \tau'$ — By Lemma 1.10 with (3)
5. $\{\} \uplus \Delta' \uplus \Delta'';\{\} \vdash \mathsf{only}\ \Delta''\ \mathsf{in}\ e : \mathsf{Ans}$ — Because by inspection of $\mathsf{htonly}\ \Delta = \Delta' \uplus \Delta''$
6. $\Delta'';\{\} \vdash e : \mathsf{Ans}$ — By (5) inversion of htonly
7. $\{\};\{\} \vdash R^{\Delta''}[e] : \mathsf{Ans}$ — By definition of $R'^{\Delta''}$

   Therefore $\vdash R^{\Delta''}[e]$ wt — By wte with (7)

case $\boxed{\mathsf{rdshalt}\ \ R[E[\mathsf{halt}^{\tau'}]] \mapsto_P \mathsf{halt}^{\tau}\ \text{where}\ E \neq [\ ]}$

1. $\vdash R[E[\mathsf{halt}^{\tau'}]]$ wt — By assumption
2. $\{\};\{\} \vdash R[E[\mathsf{halt}^{\tau'}]] : \tau$ — By (1) and inversion of wte
3. $\{\} \vdash \tau$ wf — By Lemma 1.8 with (2)
4. $\{\};\{\} \vdash \mathsf{halt}^{\tau} : \tau$ — By hthalt with (3)

   Therefore $\vdash \mathsf{halt}^{\tau}$ wt — By wte with (4)

case $\boxed{\mathsf{rdsfreer}\ \ R[\mathsf{letr}\ \rho\ \mathsf{in}\ R'[e]] \mapsto_P R[R'[e]]\ \text{where}\ \vdash R[R'[e]]\ \text{wt}}$ Trivial since by assumption $\vdash R[R'[e]]$ wt

Lemma 1.2 (Progress) *If* $\vdash P_1$ wt *then, there exists* P_2 *such that* $P_1 \mapsto_P P_2$ *or* P_2 *is an answer. i.e.* P_1 *is not stuck.*

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

Consdier the cases of Lemma 1.6

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

case $\boxed{e = E[r]\ \text{where}\ r = ((\lambda x:\tau'.e')^{\Delta'}\ v)}$ P_1 reduces to P_2 using rdspure with rdsbetav.

case $\boxed{e = E[r]\ \text{where}\ r = ((\Lambda\rho.e')[\rho'])}$ P_1 reduces to P_2 using rdspure with rdstapp.

case $\boxed{e = E[r]\ \text{where}\ r = (\mathsf{fix} f:\tau'.v')}$ P_1 reduces to P_2 using rdspure with rdsfix.

case $\boxed{e = E[r]\ \text{where}\ r = \mathsf{letr}\ \rho\ \mathsf{in}\ e'}$ P_1 reduces to P_2 using rdsletr.

case $\boxed{e = E[r]\ \text{where}\ r = \mathsf{get}[\rho](\mathsf{put}[\rho](v))}$ P_1 reduces to P_2 using rdsget.

case $\boxed{e = E[r]\ \text{where}\ r = \mathsf{only}\ \Delta'\ \mathsf{in}\ e'}$ P_1 reduces to P_2 using rdsonly.

case $\boxed{e = E[r] \text{ where } r = \mathsf{halt}^{\tau'}}$ If $E \neq [\,]$ then P_1 reduces to P_2 using rdshalt with rdsfix. If $E = [\,]$ then P_1 is an answer.

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

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

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

case $\boxed{\dfrac{}{\Delta; \Gamma \uplus \{x : \tau\} \vdash x : \tau}\ \mathsf{htvar}}$ Because $e' = x$, $\tau' = \tau$ and $x[e/x] = e$ therefore $\Delta; \Gamma \vdash e : \tau$ by assumption

case $\boxed{\dfrac{}{\Delta; \Gamma \uplus \{x : \tau\} \uplus \{y : \tau'\} \vdash y : \tau'}\ \mathsf{htvar}}$ Because $e' = y$, $y[e/x] = y$ therefore $\Delta; \Gamma \uplus \{y : \tau'\} \vdash y : \tau'$ by htvar

case $\boxed{\dfrac{}{\Delta; \Gamma \uplus \{x : \tau\} \vdash \langle\rangle : \mathsf{unit}}\ \mathsf{htunit}}$ Because $e' = \langle\rangle$, $\tau' = \mathsf{unit}$, and $\langle\rangle[e/x] = \langle\rangle$ therefore $\Delta; \Gamma \vdash \langle\rangle : \mathsf{unit}$ by htunit

case $\boxed{\dfrac{\Delta' \vdash (\Gamma' \uplus \{x : \tau\})\ \text{wfenv} \quad \Delta'' \vdash \tau_1\ \text{wf} \quad \Delta''; \Gamma'' \uplus \{y : \tau_1\} \vdash e'' : \tau_2}{\Delta' \uplus \Delta''; (\Gamma' \uplus \{x : \tau\}) \uplus \Gamma'' \vdash (\lambda y : \tau_1.e'')^{\Delta''} : \tau_1 \xrightarrow{\Delta''} \tau_2}\ \mathsf{htabs}}$ Because $\Gamma = \Gamma' \uplus \Gamma''$, $\Delta = \Delta' \uplus \Delta''$, $e' = (\lambda y : \tau_1.e'')^{\Delta''}$, $\tau' = \tau_1 \xrightarrow{\Delta''} \tau_2$, and $(\lambda y : \tau_1.e'')^{\Delta''}[e/x] = (\lambda y : \tau_1.e''[e/x])^{\Delta''}$

1. $\Delta''; \Gamma'' \uplus \{y : \tau_1\}\tau \vdash e'' : \tau_2$ By assumption
2. $e''[e/x] = e''$ Because (1) implies that x is not free in e''

   Therefore $\Delta' \uplus \Delta''; \Gamma' \uplus \Gamma'' \vdash (\lambda y : \tau_1.e'')^{\Delta''} : \tau_1 \xrightarrow{\Delta''} \tau_2$ By assumption

case $\boxed{\dfrac{\Delta' \vdash \Gamma'\ \text{wfenv} \quad \Delta'' \vdash \tau_1\ \text{wf} \quad \Delta''; (\Gamma'' \uplus \{x : \tau\}) \uplus \{y : \tau_1\} \vdash e'' : \tau_2}{\Delta' \uplus \Delta''; \Gamma' \uplus (\Gamma'' \uplus \{x : \tau\}) \vdash (\lambda y : \tau_1.e'')^{\Delta''} : \tau_1 \xrightarrow{\Delta''} \tau_2}\ \mathsf{htabs}}$ Because $\Gamma = \Gamma' \uplus \Gamma''$, $\Delta = \Delta' \uplus \Delta''$, $e' = (\lambda y : \tau_1.e'')^{\Delta''}$, $\tau' = \tau_1 \xrightarrow{\Delta''} \tau_2$, and $(\lambda y : \tau_1.e'')^{\Delta''}[e/x] = (\lambda y : \tau_1.e''[e/x])^{\Delta''}$

1. $\Delta' \uplus \Delta''; \Gamma' \uplus \Gamma'' \vdash e : \tau$ By assumption
2. $\Delta' \vdash \Gamma'$ wfenv By assumption
3. $\Delta'' \vdash \tau_1$ wf By assumption
4. $\Delta''; \Gamma'' \uplus \{y : \tau_1\} \uplus \{x : \tau\} \vdash e'' : \tau_2$ By assumption and exchange
5. $\Delta'' \vdash \Gamma''$ wfenv 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 $e = (\mathsf{fix} f : \tau'.v')$

7. $\Delta''; \Gamma'' \uplus \{y : \tau_1\} \vdash e''[e/x] : \tau_2$ — By IH with (6) and (4)

Therefore $\Delta' \uplus \Delta''; \Gamma' \uplus \Gamma'' \vdash (\lambda y : \tau_1.e''[e/x])^{\Delta''} : \tau_1 \xrightarrow{\Delta''} \tau_2$ — By htabs with (2), (3) and (7)

case $\dfrac{\Delta' \uplus \Delta''; \Gamma \uplus \{x : \tau\} \vdash e_1 : \tau_1 \xrightarrow{\Delta''} \tau_2 \quad \Delta' \uplus \Delta''; \Gamma \uplus \{x : \tau\} \vdash e_2 : \tau_1}{\Delta' \uplus \Delta''; \Gamma \uplus \{x : \tau\} \vdash (e_1\ e_2) : \tau_2}\ \mathsf{htapp}$ Because $\Delta = \Delta' \uplus \Delta''$, $e' = (e_1\ e_2)$, $\tau' = \tau_2$, and $(e_1\ e_2)[e/x] = (e_1[e/x]\ e_2[e/x])$

1. $\Delta' \uplus \Delta''; \Gamma \vdash e : \tau$ — By assumption
2. $\Delta' \uplus \Delta''; \Gamma \uplus \{x : \tau\} \vdash e_1 : \tau_1 \xrightarrow{\Delta''} \tau_2$ — By assumption
3. $\Delta' \uplus \Delta''; \Gamma \uplus \{x : \tau\} \vdash e_2 : \tau_1$ — By assumption
4. $\Delta' \uplus \Delta''; \Gamma \vdash e_1[e/x] : \tau_1 \xrightarrow{\Delta''} \tau_2$ — By IH with (1) and (2)
5. $\Delta' \uplus \Delta''; \Gamma \vdash e_2[e/x] : \tau_1$ — By IH with (1) and (3)

Therefore $\Delta' \uplus \Delta''; \Gamma \vdash (e_1[e/x]\ e_2[e/x]) : \tau_2$ — By htapp with (4) and (5)

case $\dfrac{\Delta \uplus \{\rho\}; \Gamma \uplus \{x : \tau\} \vdash e'' : \tau''}{\Delta; \Gamma \uplus \{x : \tau\} \vdash (\Lambda\rho.e'') : \forall\rho.\tau''}\ \mathsf{httabs}$ Because $e' = (\Lambda\rho.e'')$, $\tau' = \forall\rho.\tau''$, and $(\Lambda\rho.e'')[e/x] = (\Lambda\rho.e''[e/x])$

1. $\Delta; \Gamma \vdash e : \tau$ — By assumption
2. $\Delta \uplus \{\rho\}; \Gamma \uplus \{x : \tau\} \vdash e'' : \tau''$ — By assumption
3. $\Delta \uplus \{\rho\}; \Gamma \vdash e''[e/x] : \tau''$ — By IH with (1) and (2)

Therefore $\Delta; \Gamma \vdash (\Lambda\rho.e''[e/x]) : \forall\rho.\tau''$ — By httabs and (3)

case $\dfrac{\Delta \uplus \{\rho'\}; \Gamma \uplus \{x : \tau\} \vdash e'' : \forall\rho.\tau''}{\Delta \uplus \{\rho'\}; \Gamma \uplus \{x : \tau\} \vdash (e''[\rho']) : \tau''[\rho'/\rho]}\ \mathsf{httapp}$ Because $e' = (e''[\rho'])$, $\tau' = \tau''[\rho'/\rho]$, and $(e''[\rho'])[e/x] = (e''[e/x]\ \rho')$

1. $\Delta; \Gamma \vdash e : \tau$ — By assumption
2. $\Delta \vdash \rho'$ wf — By assumption
3. $\Delta; \Gamma \uplus \{x : \tau\} \vdash e'' : \forall\rho.\tau''$ — By assumption
4. $\Delta; \Gamma \vdash e''[e/x] : \forall\rho.\tau''$ — By IH with (1) and (3)

Therefore $\Delta; \Gamma \vdash (e''[e/x][\rho']) : \tau''[\rho'/\rho]$ — By httapp with (2) and (4)

case $\dfrac{\Delta \vdash \tau' \text{ wf} \quad \Delta \uplus \{\rho\}; \Gamma \uplus \{x : \tau\} \vdash e'' : \tau'}{\Delta; \Gamma \uplus \{x : \tau\} \vdash (\mathsf{letr}\ \rho\ \mathsf{in}\ e'') : \tau'}\ \mathsf{htletr}$ Because $e' = \mathsf{letr}\ \rho\ \mathsf{in}\ e''$ and $\mathsf{letr}\ \rho\ \mathsf{in}\ e''[e/x] = \mathsf{letr}\ \rho\ \mathsf{in}\ e''[e/x]$

1. $\Delta;\Gamma \vdash e:\tau$ By assumption
2. $\Delta \vdash \tau'$ wf By assumption
3. $\Delta \uplus \{\rho\};\Gamma \uplus \{x:\tau\} \vdash e'':\tau'$ By assumption
4. $\Delta \uplus \{\rho\};\Gamma \vdash e''[e/x]:\tau'$ By IH with (1) and (3)

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

case $\dfrac{\Delta' \uplus \{\rho\};\Gamma \uplus \{x:\tau\} \vdash e'':\tau''}{\Delta' \uplus \{\rho\};\Gamma \uplus \{x:\tau\} \vdash \mathsf{put}[\rho](e''):(\tau''\ \mathsf{at}\ \rho)}\ \mathsf{htput}$ Because $\Delta = \Delta' \uplus \{\rho\}$, $e' = \mathsf{put}[\rho](e'')$, $\tau' = (\tau''\ \mathsf{at}\ \rho)$, and $\mathsf{put}[\rho](e'')[e/x] = \mathsf{put}[\rho](e''[e/x])$

1. $\Delta' \uplus \{\rho\};\Gamma \vdash e:\tau$ By assumption
2. $\Delta' \uplus \{\rho\};\Gamma \uplus \{x:\tau\} \vdash e'':\tau''$ By assumption
3. $\Delta' \uplus \{\rho\};\Gamma \vdash e''[e/x]:\tau''$ By IH with (1) and (2)

   Therefore $\Delta' \uplus \{\rho\};\Gamma \vdash \mathsf{put}[\rho](e''[e/x]):(\rho\ \mathsf{at}\ \tau'')$ By htput with (3)

case $\dfrac{\Delta' \uplus \{\rho\};\Gamma \uplus \{x:\tau\} \vdash e'':(\tau'\ \mathsf{at}\ \rho)}{\Delta' \uplus \{\rho\};\Gamma \uplus \{x:\tau\} \vdash \mathsf{get}[\rho](e''):\tau'}\ \mathsf{htget}$ Because $\Delta = \Delta' \uplus \{\rho\}$, $e' = \mathsf{get}[\rho](e'')$, $\mathsf{get}[\rho](e'')[e/x] = \mathsf{get}[\rho](e''[e/x])$

1. $\Delta' \uplus \{\rho\};\Gamma \vdash e:\tau$ By assumption
2. $\Delta' \uplus \{\rho\};\Gamma \uplus \{x:\tau\} \vdash e'':(\rho\ \mathsf{at}\ \tau')$ By assumption
3. $\Delta' \uplus \{\rho\};\Gamma \vdash e''[e/x]:(\rho\ \mathsf{at}\ \tau')$ By IH with (1) and (2)

   Therefore $\Delta' \uplus \{\rho\};\Gamma \vdash \mathsf{get}[\rho](e''[e/x]):\tau'$ By htget with (3)

case $\dfrac{\Delta' \vdash \Gamma' \uplus \{x:\tau\}\ \text{wfenv} \qquad \Delta'';\Gamma'' \vdash e'':\mathsf{Ans}}{\Delta' \uplus \Delta'';\Gamma' \uplus \{x:\tau\} \uplus \Gamma'' \vdash (\mathsf{only}\ \Delta''\ \mathsf{in}\ e''):\mathsf{Ans}}\ \mathsf{htonly}$

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

1. $\Delta'';\Gamma'' \vdash e'':\mathsf{Ans}$ By assumption
2. $e''[e/x] =$ e" Because (1) implies x is not free in e''

   Therefore $\Delta' \uplus \Delta'';\Gamma' \uplus \Gamma'' \vdash \mathsf{only}\ \Delta''\ \mathsf{in}\ e'':\mathsf{Ans}$ By assumption

case $\dfrac{\Delta' \vdash \Gamma'\ \text{wfenv} \qquad \Delta'';\Gamma'' \uplus \{x:\tau\} \vdash e'':\mathsf{Ans}}{\Delta' \uplus \Delta'';\Gamma' \uplus \Gamma'' \uplus \{x:\tau\} \vdash (\mathsf{only}\ \Delta''\ \mathsf{in}\ e''):\mathsf{Ans}}\ \mathsf{htonly}$

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

1. $\Delta' \uplus \Delta'';\Gamma' \uplus \Gamma'' \vdash e:\tau$ By assumption
2. $\Delta'';\Gamma'' \uplus \{x:\tau\} \vdash e'':\mathsf{Ans}$ By assumption

3. $\Delta' \vdash \Gamma'$ wfenv — By assumption
4. $\Delta'' \vdash \Gamma'' \uplus \{x:\tau\}$ wfenv — By implicit assumption
5. $\Delta'' \vdash \tau$ wfenv — By (4) and inversion of $\mathsf{wfenvbv}$
6. $\Delta''; \Gamma'' \vdash e:\tau$ — By Lemma 1.4 with (5), (3), (4), and (1)
7. $\Delta''; \Gamma'' \vdash e''[e/x]:\mathsf{Ans}$ — By IH with (6) and (2)

Therefore $\Delta' \uplus \Delta''; \Gamma' \uplus \Gamma'' \vdash \mathsf{only}\ \Delta''\ \mathsf{in}\ e''[e/x]:\mathsf{Ans}$ By htonly with (3) and (7)

case $\boxed{\dfrac{\Delta \vdash \tau' \text{ wf} \quad \Delta; \Gamma \uplus \{\tau:x\} \uplus \{f:\tau'\} \vdash v:\tau'}{\Delta; \Gamma \uplus \{\tau:x\} \vdash (\mathsf{fix} f:\tau'.v):\tau'}\ \mathsf{htfix}}$ Because $e' = (\mathsf{fix} f:\tau'.v)$ and $(\mathsf{fix} f:\tau'.v)[e/x] = (\mathsf{fix} f:\tau'.v[e/x])$

1. $\Delta; \Gamma \vdash e:\tau$ — By assumption
2. $\Delta \vdash \tau'$ wf — By assumption
3. $\Delta; \Gamma \uplus \{f:\tau'\} \uplus \{x:\tau\} \vdash v:\tau'$ — By assumption of htfix
4. $\Delta; \Gamma \uplus \{f:\tau'\} \vdash v[e/x]:\tau'$ — By IH with (1) and (3)

Therefore $\Delta; \Gamma \vdash (\mathsf{fix} f:\tau'.v[e/x]):\tau'$ By htfix with (2) and (4)

case $\boxed{\dfrac{\Delta \vdash \tau' \text{ wf}}{\Delta; \Gamma \uplus \{x:\tau\} \vdash \mathsf{halt}^{\tau'}:\tau'}\ \mathsf{hthalt}}$ Because $e' = \mathsf{halt}^{\tau'}$ and $\mathsf{halt}^{\tau'}[e/x] = \mathsf{halt}^{\tau'}$ by assumption $\Delta \vdash \tau'$ wf therefore $\Delta; \Gamma \vdash \mathsf{halt}^{\tau'}:\tau'$ by hthalt

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

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

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

case $\boxed{\dfrac{}{\Delta \uplus \Delta'; \Gamma \uplus \Gamma' \vdash \langle\rangle:\mathsf{unit}}\ \mathsf{htunit}}$ Trivial since $\Delta' \vdash \Gamma'$ wfenv therefore $\Delta'; \Gamma' \vdash \langle\rangle:\mathsf{unit}$ by htunit

case $\boxed{\dfrac{\Delta \vdash \Gamma' \text{ wfenv} \quad \Delta' \vdash \tau_1 \text{ wf} \quad \Delta'; \Gamma \uplus \{x:\tau_1\} \vdash e':\tau_2}{\Delta \uplus \Delta'; \Gamma' \uplus \Gamma \vdash (\lambda x:\tau_1.e')^{\Delta'}:\tau_1 \xrightarrow{\Delta'} \tau_2}\ \mathsf{htabs}}$ Because $\tau = \tau_1 \xrightarrow{\Delta'} \tau_2$,

1. $\Delta' \vdash \tau_1$ wf — By assumption
2. $\Delta'; \Gamma' \uplus \{x:\tau_1\} \vdash e':\tau_2$ — By assumption
3. $\{\} \vdash \{\}$ wfenv — By $\mathsf{wfenvenpty}$

Therefore $\{\} \uplus \Delta'; \{\} \uplus \Gamma' \vdash (\lambda x:\tau_1.e')^{\Delta'}:\tau_1 \xrightarrow{\Delta'} \tau_2$ By htabs with (3), (1), and (2)

case $\boxed{\dfrac{\Delta \uplus \Delta' \uplus \{\rho\}; \Gamma \uplus \Gamma' \vdash e' : \tau'}{\Delta \uplus \Delta'; \Gamma \uplus \Gamma' \vdash (\Lambda\rho.e') : \forall\rho.\tau'}\ \mathsf{httabs}}$

1. $\Delta' \vdash \forall\rho.\tau'$ wf — By assumption
2. $\Delta \vdash \Gamma$ wfenv — By assumption
3. $\Delta' \vdash \Gamma'$ wfenv — By assumption
4. $\Delta \uplus \Delta' \uplus \{\rho\}; \Gamma \uplus \Gamma' \vdash e' : \tau'$ — By assumption
5. $\Delta' \uplus \{\rho\} \vdash \tau'$ wf — By (1) and inversion of wfall
6. $\Delta' \uplus \{\rho\} \vdash \Gamma'$ wfenv — By (3) and weakening
7. $\Delta' \uplus \{\rho\}; \Gamma' \vdash e' : \tau'$ — By IH with (5), (2), (6), and (4)

Therefore $\Delta'; \Gamma' \vdash (\Lambda\rho.e') : \forall\rho.\tau'$ — By httabs with (7)

case $\boxed{\dfrac{\Delta \uplus \Delta'' \uplus \{\rho\}; \Gamma \uplus \Gamma' \vdash v'' : \tau'}{\Delta \uplus \Delta'' \uplus \{\rho\}; \Gamma \uplus \Gamma' \vdash \mathsf{put}[\rho](v'') : (\tau'\ \mathsf{at}\ \rho)}\ \mathsf{htput}}$ Because $\Delta' = \Delta'' \uplus \{\rho\}$ and $\tau = (\tau'\ \mathsf{at}\ \rho)$

1. $\Delta'' \uplus \{\rho\} \vdash (\tau'\ \mathsf{at}\ \rho)$ wf — By assumption
2. $\Delta \vdash \Gamma$ wfenv — By assumption
3. $\Delta'' \uplus \{\rho\} \vdash \Gamma'$ wfenv — By assumption
4. $\Delta \uplus \Delta'' \uplus \{\rho\}; \Gamma \uplus \Gamma' \vdash v' : \tau'$ — By assumption
5. $\Delta'' \uplus \{\rho\} \vdash \tau'$ wf — By (1) and inversion of wfat
6. $\Delta'' \uplus \{\rho\}; \Gamma' \vdash v' : \tau'$ — By IH with (5), (3), (2), and (4)

Therefore $\Delta'' \uplus \{\rho\}; \Gamma' \vdash \mathsf{put}[\rho](v') : (\tau'\ \mathsf{at}\ \rho)$ — By htput with (6)

case $\boxed{\dfrac{\Delta \uplus \Delta' \vdash \tau \text{ wf} \quad \Delta \uplus \Delta'; \Gamma \uplus \Gamma' \uplus \{f : \tau\} \vdash v' : \tau}{\Delta \uplus \Delta'; \Gamma \uplus \Gamma' \vdash (\mathsf{fix} f : \tau.v') : \tau}\ \mathsf{htfix}}$

1. $\Delta' \vdash \tau$ wf — By assumption
2. $\Delta \vdash \Gamma$ wfenv — By assumption
3. $\Delta' \vdash \Gamma'$ wfenv — By assumption
4. $\Delta' \vdash \Gamma' \uplus \{f : \tau\}$ wfenv — By wfenvbv with (3) and (1)
5. $\Delta \uplus \Delta'; \Gamma \uplus \Gamma' \uplus \{f : \tau\} \vdash v' : \tau$ — By assumption
6. $\Delta'; \Gamma' \uplus \{f : \tau\} \vdash v' : \tau$ — By IH with (1), (2), (4) and (5)

Therefore $\Delta'; \Gamma' \vdash (\mathsf{fix} f : \tau.v') : \tau$ — By htfix with (1) and (6)

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 $e_1 \mapsto_e e_2$

Consdier the cases of $e_1 \mapsto_e e_2$

case $\boxed{\mathsf{rdsbetav} \quad ((\lambda x{:}\tau_1.e)^{\Delta''}\ v) \mapsto_e e[v/x]}$ Because $\Delta = \Delta' \uplus \Delta''$ and $\{\} = \{\} \uplus \{\}$

1. $\Delta' \uplus \Delta''; \{\} \uplus \{\} \vdash ((\lambda x{:}\tau_1.e)^{\Delta''}\ v){:}\tau_2$ — By assumption
2. $\Delta' \uplus \Delta''; \{\} \uplus \{\} \vdash (\lambda x{:}\tau_1.e)^{\Delta''}{:}\tau_1 \xrightarrow{\Delta''} \tau_2$ — By (1) and inversion of htapp
3. $\Delta' \uplus \Delta''; \{\} \uplus \{\} \vdash v{:}\tau_1$ — By (1) and inversion of htapp
4. $\Delta''; \{\} \uplus \{x{:}\tau_1\} \vdash e{:}\tau_2$ — By (2) and inversion of htabs
5. $\Delta' \uplus \Delta''; \{\} \uplus \{\} \uplus \{x{:}\tau_1\} \vdash e{:}\tau_2$ — By (4) and weakening

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

case $\boxed{\mathsf{rdstapp} \quad ((\Lambda\rho.e)[\rho']) \mapsto_e e[\rho'/\rho]}$ Because $\Delta = \Delta' \uplus \{\rho'\}$

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

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

case $\boxed{\mathsf{rdsfix} \quad (\mathsf{fix} f{:}\tau.v) \mapsto_e v[(\mathsf{fix} f{:}\tau.v)/f]}$

1. $\Delta; \{\} \vdash (\mathsf{fix} f{:}\tau.v){:}\tau$ — By assumption
2. $\Delta; \{\} \uplus \{f{:}\tau\} \vdash v{:}\tau$ — By (1) and inversion of htfix

   It follows that $\Delta; \{\} \vdash v[(\mathsf{fix} f{:}\tau.v)/f]{:}\tau$ — 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. $((\lambda x{:}\tau'.e')^{\Delta''}\ v)$ *where* $\Delta = \Delta' \uplus \Delta''$
2. $((\Lambda\rho.e')[\tau'])$
3. $(\mathsf{fix} f{:}\tau'.v)$
4. $\mathsf{letr}\ \rho\ \mathsf{in}\ e'$
5. $\mathsf{get}[\rho](\mathsf{put}[\rho](e'))$
6. $\mathsf{only}\ \Delta'\ \mathsf{in}\ e'$
7. $\mathsf{halt}^{\tau'}$

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 $\boxed{e = \langle\rangle}$ e is a value

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Proof. By inspection of the typing judgments for τ

case $\boxed{\tau = \mathsf{unit}}$ Follows from inversion of htunit

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

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

case $\boxed{\tau = (\tau' \ \mathsf{at}\ \rho)}$ Follows from inversion of 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 $\boxed{\dfrac{}{\Delta; \Gamma \uplus \{x : \tau\} \vdash x : \tau}\ \mathsf{htvar}}$ $\Delta \vdash \Gamma \uplus \{x : \tau\}$ wf by implicit assumption. Therefore $\Delta \vdash \tau$ wf by $\Delta \vdash \Gamma \uplus \{x : \tau\}$ wf and inversion of $\mathsf{wfenvbv}$

case $\boxed{\dfrac{}{\Delta; \Gamma \vdash \langle\rangle : \mathsf{unit}}\ \mathsf{htunit}}$ $\Delta \vdash \mathsf{unit}$ wf by wfunit

case $\boxed{\dfrac{\Delta' \vdash \Gamma' \text{ wfenv} \quad \Delta'' \vdash \tau_1 \text{ wf} \quad \Delta''; \Gamma'' \uplus \{x : \tau_1\} \vdash e' : \tau_2}{\Delta' \uplus \Delta''; \Gamma' \uplus \Gamma'' \vdash (\lambda x : \tau_1.e')^{\Delta''} : \tau_1 \xrightarrow{\Delta''} \tau_2}\ \mathsf{htabs}}$ 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 $\mathsf{wfarrow}$ with (1) and (3)

   Therefore $\Delta' \uplus \Delta'' \vdash \tau_1 \xrightarrow{\Delta''} \tau_2$ wf — By weakening

case $\boxed{\dfrac{\Delta' \uplus \Delta''; \Gamma \vdash e_1 : \tau_1 \xrightarrow{\Delta''} \tau_2 \quad \Delta' \uplus \Delta''; \Gamma \vdash e_2 : \tau_1}{\Delta' \uplus \Delta''; \Gamma \vdash (e_1\ e_2) : \tau_2}\ \mathsf{htapp}}$ Because $\Delta = \Delta' \uplus \Delta''$ and $\Gamma = \Gamma' \uplus \Gamma''$

1. $\Delta' \uplus \Delta''; \Gamma' \uplus \Gamma'' \vdash e_1 : \tau_1 \xrightarrow{\Delta'} \tau_2$ — By assumption
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 $\mathsf{wfarrow}$

case $\boxed{\dfrac{\Delta \uplus \{\rho\}; \Gamma \vdash e' : \tau'}{\Delta; \Gamma \vdash (\Lambda\rho.e') : \forall\rho.\tau'}\ \mathsf{httabs}}$

1. $\Delta \uplus \{\rho\}; \Gamma \vdash e' : \tau'$ — By assumption
2. $\Delta \uplus \{\rho\} \vdash \tau'$ wf — By IH with (1)

Therefore $\Delta \vdash \forall\rho.\tau'$ wf — By wfall with (2)

case $\boxed{\dfrac{\Delta' \uplus \{\rho\}; \Gamma \vdash \rho' : \forall e'.\tau'}{\Delta' \uplus \{\rho\}; \Gamma \vdash (\rho'[\rho]) : \tau'[\rho/e']}\ \mathsf{httapp}}$ Because $\Delta = \Delta' \uplus \{\rho'\}$

1. $\Delta' \uplus \{\rho'\}; \Gamma \vdash e' : \forall\rho.\tau'$ — By assumption
2. $\Delta' \uplus \{\rho'\} \vdash \forall\rho.\tau'$ wf — By IH with (1)
3. $\Delta' \uplus \{\rho'\} \uplus \{\rho\} \vdash \tau'$ wf — By (2) and inversion of wfall

Therefore $\Delta' \uplus \{\rho'\} \vdash \tau'[\rho'/\rho]$ wf — By induction on the derivations of $\Delta' \uplus \{\rho'\} \uplus \{\rho\} \vdash \tau'$ wf and (3)

case $\boxed{\dfrac{\Delta \vdash \tau \text{ wf} \quad \Delta \uplus \{\rho\}; \Gamma \vdash e' : \tau}{\Delta; \Gamma \vdash (\mathsf{letr}\ \rho\ \mathsf{in}\ e') : \tau}\ \mathsf{htletr}}$ $\Delta \vdash \tau$ wf by assumption

case $\boxed{\dfrac{\Delta' \uplus \{\rho\}; \Gamma \vdash e' : \tau'}{\Delta' \uplus \{\rho\}; \Gamma \vdash \mathsf{put}[\rho](e') : (\tau'\ \mathsf{at}\ \rho)}\ \mathsf{htput}}$ Because $\Delta = \Delta' \uplus \{\rho\}$

1. $\Delta' \uplus \{\rho\} \vdash e' : \tau'$ — By assumption
2. $\Delta' \uplus \{\rho\} \vdash \tau'$ wf — By IH with (1)

Therefore $\Delta' \uplus \{\rho\} \vdash (\tau'\ \mathsf{at}\ \rho)$ wf — By wfat with (2)

case $\boxed{\dfrac{\Delta' \uplus \{\rho\}; \Gamma \vdash e' : (\tau'\ \mathsf{at}\ \rho)}{\Delta' \uplus \{\rho\}; \Gamma \vdash \mathsf{get}[\rho](e') : \tau'}\ \mathsf{htget}}$ Because $\Delta = \Delta' \uplus \{\rho\}$

1. $\Delta' \uplus \{\rho\} \vdash e' : (\tau'\ \mathsf{at}\ \rho)$ — By assumption
2. $\Delta' \uplus \{\rho\} \vdash (\tau'\ \mathsf{at}\ \rho)$ wf — By IH with (1)

Therefore $\Delta' \uplus \{\rho\} \vdash \tau'$ wf — By (2) and inversion of wfat

case $\boxed{\dfrac{\Delta' \vdash \Gamma' \text{ wfenv} \qquad \Delta''; \Gamma'' \vdash e' : \mathsf{Ans}}{\Delta' \uplus \Delta''; \Gamma' \uplus \Gamma'' \vdash (\mathsf{only}\ \Delta''\ \mathsf{in}\ e') : \mathsf{Ans}}\ \mathsf{htonly}}$ Because $\Delta = \Delta' \uplus \Delta''$, $\Delta' \uplus \Delta'' \vdash \mathsf{Ans}$ wf by wfAns

case $\boxed{\dfrac{\Delta \vdash \tau \text{ wf} \quad \Delta; \Gamma \uplus \{f : \tau\} \vdash v : \tau}{\Delta; \Gamma \vdash (\mathsf{fix} f : \tau.v) : \tau}\ \mathsf{htfix}}$ $\Delta \vdash \tau$ wf by assumption

case $\boxed{\dfrac{\Delta \vdash \tau \text{ wf}}{\Delta;\Gamma \vdash \mathsf{halt}^{\tau} : \tau}\ \mathsf{hthalt}}$ $\Delta \vdash \tau$ wf by assumption

Lemma 1.9 (Typing Under Region Variable Subsitution) *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* $\Delta; \Gamma \vdash e : \tau'$.

Proof. By induction on the structure of E

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

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

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

1. $\Delta' \uplus \Delta''; \Gamma \vdash (E'[e]\ e') : \tau$ — By assumption
2. $\Delta' \uplus \Delta''; \Gamma \vdash E'[e] : \tau_1 \xrightarrow{\Delta''} \tau$ — By (1) and inversion of htapp

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

case $\boxed{E = (v\ E')}$ 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 $\boxed{E = (E'[\rho'])}$ 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 httapp

   Therefore $\Delta; \Gamma \vdash e : \tau'$ — By IH and (2)

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

1. $\Delta' \uplus \{\rho\}; \Gamma \vdash \mathsf{put}[\rho](E'[e]) : \tau$ — 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 $\boxed{E = \mathsf{get}[\rho](E')}$ Because $E[e] = (\rho\ E'[e])$ and $\Delta = \Delta' \uplus \{\rho\}$

1. $\Delta' \uplus \{\rho\}; \Gamma \vdash \mathsf{get}[\rho](E'[e]) : \tau$ 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 $\boxed{E = [\,]}$ Because $E[e_2] = e_2$ and $\tau = \tau'$ therefore $\Delta; \Gamma \vdash e_2 : \tau$ by assumption

case $\boxed{E = (E'\ e')}$ 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. $\Delta' \uplus \Delta''; \Gamma \vdash E'[e_1] : \tau_1 \xrightarrow{\Delta''} \tau'$ By (3) and inversion of htapp
5. $\Delta' \uplus \Delta''; \Gamma \vdash e' : \tau_1$ By (3) and inversion of htapp
6. $\Delta' \uplus \Delta''; \Gamma \vdash E'[e_2] : \tau_1 \xrightarrow{\Delta''} \tau'$ 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 $\boxed{E = (v\ E')}$ 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. $\Delta' \uplus \Delta''; \Gamma \vdash v : \tau_1 \xrightarrow{\Delta''} \tau'$ 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 $\boxed{E = (E'[\rho'])}$ 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 httapp

5. $\Delta' \uplus \{\rho'\}; \Gamma \vdash E'[e_2] : \forall \rho.\tau''$ By IH with (1), (2), and (4)

Therefore $\Delta' \uplus \{\rho'\}; \Gamma \vdash (E'[e_2][\rho']) : \tau'$ By httapp with (5)

case $\boxed{E = \mathsf{put}[\rho](E')}$ Because $E[e_1] = \mathsf{put}[\rho](E'[e_1])$ 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. $\Delta' \uplus \{\rho\}; \Gamma \vdash \mathsf{put}[\rho](E'[e_1]) : \tau'$ By assumption
4. $\Delta' \uplus \{\rho\}; \Gamma \vdash E'[e_1] : \tau''$ By (3) and inversion of of htput
5. $\Delta' \uplus \{\rho\}; \Gamma \vdash E'[e_2] : \tau''$ By IH with (1), (2), and (4)

Therefore $\Delta' \uplus \{\rho\}; \Gamma \vdash \mathsf{put}[\rho](E'[e_2]) : \tau'$ By htput with (5)

case $\boxed{E = \mathsf{get}[\rho](E')}$ $E[e_1] = \mathsf{get}[\rho](E'[e_1])$ 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. $\Delta' \uplus \{\rho\}; \Gamma \vdash \mathsf{get}[\rho](E'[e_1]) : \tau'$ By assumption
4. $\Delta' \uplus \{\rho\}; \Gamma \vdash E'[e_1] : \tau''$ By (3) and inversion of of htget
5. $\Delta' \uplus \{\rho\}; \Gamma \vdash E'[e_2] : \tau''$ By IH with (1), (2), and (4)

Therefore $\Delta' \uplus \{\rho\}; \Gamma \vdash \mathsf{get}[\rho](E'[e_2]) : \tau'$ By htget with (5)

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

case $\boxed{R = [\,]}$ Because $R[e] = e$ and $\Delta = \Delta \uplus \{\}$ therefore $\Delta; \Gamma \vdash e : \tau$ by assumption

case $\boxed{R = \mathsf{letr}\ \rho\ \mathsf{in}\ R'}$ Because $R[e] = \mathsf{letr}\ \rho\ \mathsf{in}\ R'[e]$

1. $\Delta; \Gamma \vdash \mathsf{letr}\ \rho\ \mathsf{in}\ R'[e] : \tau$ 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 Δ' 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 Δ' 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 $\boxed{R = [\,]}$ When $\Delta' = \{\}$ because $R[e_2] = e_2$ and $\Delta \uplus \Delta' = \Delta$ therefore $\Delta; \Gamma \vdash e_2 : \tau$ by assumption

case $\boxed{R = \mathsf{letr}\ \rho\ \mathsf{in}\ R'}$ When $\Delta' = \{\rho\}$ because $R[e_1] = \mathsf{letr}\ \rho\ \mathsf{in}\ R'[e_1]$

1. $\Delta \uplus \{\rho\}; \Gamma \vdash e_1 : \tau$ — By assumption
2. $\Delta \uplus \{\rho\}; \Gamma \vdash e_2 : \tau$ — By assumption
3. $\Delta; \Gamma \vdash \mathsf{letr}\ \rho\ \mathsf{in}\ R'[e_1] : \tau$ — By assumption
4. $\Delta \vdash \tau$ wf — By (3) and inversion of htletr
5. $\Delta \uplus \{\rho\}; \Gamma \vdash R'[e_1] : \tau$ — By (3) and inversion of htletr
6. $\Delta \uplus \{\rho\} \uplus \Delta''; \Gamma \vdash e_1 : \tau$ — By Lemma 1.12 with (1)
7. $\Delta \uplus \{\rho\} \uplus \Delta''; \Gamma \vdash e_2 : \tau$ — By Lemma 1.12 with (2)
8. $\Delta \uplus \{\rho\}; \Gamma \vdash R'[e_2] : \tau$ — By IH with (6), (7), and (3)

   Therefore $\Delta; \Gamma \vdash \mathsf{letr}\ \rho\ \mathsf{in}\ R'[e_2] : \tau$ — By htletr with (4) and (8)

Syntax

$$\begin{array}{rrcl} \textit{types} & \tau & ::= & \ldots \mid \mathsf{bool}(b) \mid \mathsf{bool} \\ \textit{terms} & e & ::= & \ldots \mid b \mid \mathsf{isabool}(e) \mid (\mathsf{if}\ e_1\ e_2\ e_3) \\ \textit{values} & v & ::= & \ldots \mid b \mid \mathsf{isabool}(b) \\ \textit{booleans} & b & ::= & \mathsf{true} \mid \mathsf{false} \\ \textit{control contexts} & E & ::= & \ldots \mid \mathsf{isabool}(E) \mid (\mathsf{if}\ E\ e_1\ e_2) \end{array}$$

Expression Reductions

$$\begin{array}{lrcl} \mathsf{rdsiftrue} & (\mathsf{if}\ \mathsf{isabool}(\mathsf{true})\ e_1\ e_2) & \mapsto_e & e_1 \\ \mathsf{rdsiffalse} & (\mathsf{if}\ \mathsf{isabool}(\mathsf{false})\ e_1\ e_2) & \mapsto_e & e_2 \end{array}$$

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 $\mathsf{bool}$and $\mathsf{bool}(b)$ the intuitive subtyping relationship is

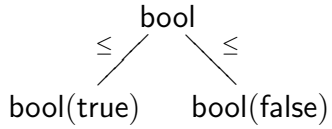

Figure 23 describes the needed extensions to our typing relations. Rather than adding a subsumption rule to our typing relation we introduce a coercion term $\mathsf{isabool}(e)$ which promotes an expression of type $\mathsf{bool}(b)$ into an expression of type bool. Note that the values of type bool are of the form $\mathsf{isabool}(b)$ while a value of type $\mathsf{bool}(b)$ is of the form b. The if expression is defined over values of type bool.

$\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\ldots$$

$$\frac{}{\Delta; \Gamma \vdash b : \mathsf{bool}(b)}\ \mathsf{htbool} \qquad \frac{\Delta; \Gamma \vdash e : \mathsf{bool}(b)}{\Delta; \Gamma \vdash \mathsf{isabool}(e) : \mathsf{bool}}\ \mathsf{htisabool}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \mathsf{bool} \quad \Delta; \Gamma \vdash e_2 : \tau \quad \Delta; \Gamma \vdash e_3 : \tau}{\Delta; \Gamma \vdash (\mathsf{if}\ e_1\ e_2\ e_3) : \tau}\ \mathsf{htif}$$

$\boxed{\Delta \vdash \tau\ \mathrm{wf}}$

$$\ldots$$

$$\frac{}{\Delta \vdash \mathsf{bool}\ \mathrm{wf}}\ \mathsf{wfbool} \qquad \frac{}{\Delta \vdash \mathsf{bool}(b)\ \mathrm{wf}}\ \mathsf{wfibool}$$

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 τ allocated in region ρ. Rather than introducing a new syntactic category for locations, we simply use variables bound by a new expression letl. Our semantics does not support mutual recursion, but by using 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 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 ρ. 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 letr bindings. To model references we extend region stacks so that between letr bindings there exists a possibly empty sequence of nested letl bindings which encodes a mapping between locations and values. We redefine the meaning of R^{Δ} appropriately. We also introduce the notation $R^{\rho,H,x,\tau,v,H'}$ which represents a region stack containing a bound region, ρ, 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 rdsderef dereferences locations. The rule rdsletupd updates and existing binding. The rule rdsletl lifts' 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 letl to encode locations we avoid the need for a separate heap type, and can type locations using the standard 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 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 ρ_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 Δ. The reduction rule rdsgclemma is safe if and only if x' does not occur in the body of e. We can be assured of this fact when ρ_1 is not in Δ. If x' did occur in e and ρ_1 is not in Δ 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 ρ_1 statically not be a member of $\Delta_2 \uplus \{\rho_2\}$, 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

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

Program Reductions

$$
\begin{array}{rcl}
[\,]^{\Delta} & \stackrel{def}{=} & [\,] \\
(\mathsf{letr}\ \rho\ \mathsf{in}\ H)^{\Delta \uplus \{\rho\}} & \stackrel{def}{=} & (\mathsf{letr}\ \rho\ \mathsf{in}\ H^{\Delta \uplus \{\rho\}}) \\
(\mathsf{letr}\ \rho\ \mathsf{in}\ H)^{\Delta} & \stackrel{def}{=} & H^{\Delta} \text{ where } \rho \notin \Delta \\
(\mathsf{letl}\ x\!:\!(\tau \ \mathsf{ref}\ \rho) = H\ \mathsf{in}\ )^{\Delta \uplus \{\rho\}} & \stackrel{def}{=} & (\mathsf{letl}\ x\!:\!(\rho \ \mathsf{ref}\ \tau) = H^{\Delta \uplus \{\rho\}}\ \mathsf{in}\ ) \\
(\mathsf{letl}\ x\!:\!(\tau \ \mathsf{ref}\ \rho) = H\ \mathsf{in}\ )^{\Delta} & \stackrel{def}{=} & H^{\Delta} \text{ where } \rho \notin \Delta
\end{array}
$$

$$
R^{\rho,H,x,\tau,v,H'} \stackrel{def}{=} R[\mathsf{letr}\ \rho\ \mathsf{in}\ H[\mathsf{letl}\ x\!:\!(\tau \ \mathsf{ref}\ \rho) = v\ \mathsf{in}\ H']]
$$

$$
\begin{array}{lrcl}
\mathsf{rdsderef} & R^{\rho,H,x,\tau,v,H'}[E[\mathsf{deref}[\rho](x)]] & \mapsto_P & R^{\rho,H,x,\tau,v,H'}[E[x]] \\
\mathsf{rdsletupd} & R^{\rho,H,x,\tau,v,H'}[E[\mathsf{update}[\rho]\ x := v'; e]] & \mapsto_P & R^{\rho,H,x,\tau,v',H'}[E[e]] \\
\mathsf{rdsletl} & R[\mathsf{letr}\ \rho\ \mathsf{in}\ H[R'[E[\mathsf{letl}\ x\!:\!(\rho \ \mathsf{ref}\ \tau) = v\ \mathsf{in}\ e]]]] & \mapsto_P & R^{\rho,H,x,\tau,v,R'}[E[e]]
\end{array}
$$

Figure 24: Regions and References

$\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\ldots$$

$$\frac{\Delta \vdash (\tau_1 \;\mathsf{ref}\; \rho)\ \mathrm{wf} \quad \Delta; \Gamma \uplus \{x : (\tau_1 \;\mathsf{ref}\; \rho)\} \vdash e_1 : \tau_1 \quad \Delta; \Gamma \uplus \{x : (\tau_1 \;\mathsf{ref}\; \rho)\} \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash (\mathsf{letl}\ x : (\tau_1 \;\mathsf{ref}\; \rho) = e_1\ \mathsf{in}\ e_2) : \tau_2}\ \mathsf{htletl}$$

$$\frac{\Delta \uplus \{\rho\}; \Gamma \vdash e : (\tau \;\mathsf{ref}\; \rho)}{\Delta \uplus \{\rho\}; \Gamma \vdash \mathsf{deref}[\rho](e) : \tau}\ \mathsf{htderef}$$

$$\frac{\Delta \uplus \{\rho\}; \Gamma \vdash e_1 : (\tau_1 \;\mathsf{ref}\; \rho) \quad \Delta \uplus \{\rho\}; \Gamma \vdash e_2 : \tau_1 \quad \Delta \uplus \{\rho\}; \Gamma \vdash e_3 : \tau_2}{\Delta \uplus \{\rho\}; \Gamma \vdash (\mathsf{update}[\rho]\ e_1 := e_2; e_3) : \tau_2}\ \mathsf{htletupd}$$

$\boxed{\Delta \vdash \tau\ \mathrm{wf}}$

$$\ldots$$

$$\frac{\Delta \uplus \{\rho\} \vdash \tau\ \mathrm{wf}}{\Delta \uplus \{\rho\} \vdash (\tau \;\mathsf{ref}\; \rho)\ \mathrm{wf}}\ \mathsf{wfref}$$

Figure 25: Typing Judgements for Regions and References

Syntax

$$
\begin{array}{rrcl}
\textit{terms} & e & ::= & \ldots \\
& & | & \mathsf{gclemma} \\
& & & \quad \mathsf{assume}\ (\mathsf{bool}(b)\ \mathsf{ref}\ \rho_1)\ \mathsf{isa}\ (\mathsf{bool\ ref}\ \rho_2)\ \mathsf{in} \\
& & & \quad \mathsf{let}\ x = e_1\ \mathsf{in} \\
& & & \quad \mathsf{only}\ \Delta\ \mathsf{in}\ e_2 \\
\textit{control contexts} & E & ::= & \ldots \\
& & | & \mathsf{gclemma} \\
& & & \quad \mathsf{assume}\ (\mathsf{bool}(b)\ \mathsf{ref}\ \rho_1)\ \mathsf{isa}\ (\mathsf{bool\ ref}\ \rho_2)\ \mathsf{in} \\
& & & \quad \mathsf{let}\ x = E\ \mathsf{in} \\
& & & \quad \mathsf{only}\ \Delta\ \mathsf{in}\ e
\end{array}
$$

$$
\begin{array}{rl}
\mathsf{rdsgclemma} & R^{\rho_1,H,x',\mathsf{bool}(b),b,H'}[E[\mathsf{gclemma} \\
& \qquad \mathsf{assume}\ (\mathsf{bool}(b)\ \mathsf{ref}\ \rho_1)\ \mathsf{isa}\ (\mathsf{bool\ ref}\ \rho_2)\ \mathsf{in} \\
& \qquad \mathsf{let}\ x = x'\ \mathsf{in} \\
& \qquad \mathsf{only}\ \Delta\ \mathsf{in}\ e]] \mapsto_P R'[E[\mathsf{only}\ \Delta\ \mathsf{in}\ e[x'/x]]] \\
& \text{where } R' = R^{\rho_1,H,x',\mathsf{bool},\mathsf{isabool}(b),H'}[\rho_2/\rho_1] \text{ and } \rho_1 \notin \Delta \\
\mathsf{rdsgclemmaerr} & R[E[\mathsf{gclemma} \\
& \quad \mathsf{assume}\ (\mathsf{bool}(b)\ \mathsf{ref}\ \rho_1)\ \mathsf{isa}\ (\mathsf{bool\ ref}\ \rho_2)\ \mathsf{in} \\
& \quad \mathsf{let}\ x = v\ \mathsf{in} \\
& \quad \mathsf{only}\ \Delta \uplus \{\rho_1\}\ \mathsf{in}\ e]] \mapsto_P \mathsf{halt}^{\mathsf{Ans}}
\end{array}
$$

Figure 26: Covariant References

$\boxed{\Delta;\Gamma \vdash e:\tau}$

$$\ldots$$

$$
\frac{\begin{array}{rl}
\Delta_1 \uplus \{\rho_1\} & \vdash \Gamma_1\ \text{wfenv} \\
\Delta_1 \uplus \{\rho_1\} \uplus \Delta_2; \Gamma_1 \uplus \Gamma_2 & \vdash e_1 : (\mathsf{bool}(b)\ \mathsf{ref}\ \rho_1) \\
\Delta_2 \uplus \{\rho_2\}; \Gamma_2 \uplus \{x : (\mathsf{bool\ ref}\ \rho_2)\} & \vdash e_2 : \mathsf{Ans}
\end{array}}{\begin{array}{l}
\Delta_1 \uplus \{\rho_1\} \uplus \Delta_2; \Gamma_1 \uplus \Gamma_2 \vdash (\mathsf{gclemma} \\
\qquad \mathsf{assume}\ (\mathsf{bool}(b)\ \mathsf{ref}\ \rho_1)\ \mathsf{isa}\ (\mathsf{bool\ ref}\ \rho_2)\ \mathsf{in} \\
\qquad \mathsf{let}\ x = e_1\ \mathsf{in} \\
\qquad \mathsf{only}\ \Delta_2 \uplus \{\rho_2\}\ \mathsf{in}\ e_2) : \mathsf{Ans}
\end{array}}\ \mathsf{htgclemma}
$$

Figure 27: Typing Judgements for Covariant References

our computation using the rdsgclemmaerr rule.