Garbage Collection Safety for Region-based Memory Management

Martin Elsman[*]
mael@it.edu

IT University of Copenhagen, Denmark
Glentevej 67, DK-2400 Copenhagen NV, Denmark.

ABSTRACT

In this paper, we prove the safety of integrating region-based memory management and Cheney-style copying garbage collection. The safety property relies on a refinement of the region typing rules that forbids dangling pointers during evaluation.

To accommodate the triggering of garbage collection at any step in the evaluation process, we base our type-safety result for the region-based system on a small-step contextual semantics and show that whenever a well-typed expression reduces to another expression, possibly by deallocating a region, then no dangling pointer is introduced. Because there are no dangling pointers in the initial heap, no dangling pointers appear during evaluation.

Although in principle, the refinement of the region typing rules leads to less flexibility and can cause worse memory behavior than when dangling pointers are permitted, experiments show that, for a range of benchmark programs, the refinement has little effect on overall memory behavior.

Categories and Subject Descriptors

D.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3 [**Programming Languages**]: Language Constructs and Features—*Dynamic storage management*; F.3 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Performance, Languages, Theory

Keywords

Garbage collection, region inference, Standard ML

[*]Part time at Royal Veterinary and Agricultural University, Denmark.

1. INTRODUCTION

In previous work, Hallenberg et al. [4] have integrated region-based memory management with a Cheney-style copying garbage collection algorithm. The algorithm is implemented for all of Standard ML in the ML Kit compiler [10]. Measurements show that, for programs that are not optimized for regions, adding garbage collection reduces memory usage. From the point of view of garbage collection, measurements demonstrate that the pressure on the garbage collector is reduced significantly by integrating garbage collection with region inference.

Much attention has been focussed on proving type safety of region inference [12, 5, 2, 3]. At runtime, the store is organized as a stack of dynamically expandable regions. Region inference is the process of inserting allocation and deallocation directives in the program at compile time. Value-creating expressions are annotated with information that controls in what region values go at runtime. Moreover, if e is some region-annotated expression, then so is

$$\texttt{letregion}\ \rho\ \texttt{in}\ e\ \texttt{end}$$

An expression of this form is evaluated by first allocating a new region on top of the region stack and binding the region to the region variable ρ. Then e is evaluated, possibly using the region bound to ρ for holding values. Finally, upon reaching `end`, the region is reclaimed.

Region inference allows for both *shallow* and *deep* pointers, that is, pointers from older regions to newer regions and from newer regions to older regions. A shallow pointer may turn into a dangling pointer if the newer region is deallocated before the older region [11]. All previous work on proving type safety of region inference focuses on showing that when a region is deallocated, none of the values in the region is used in the remainder of the computation. In particular, if region deallocation introduces dangling pointers, then none of these pointers is dereferenced in the remainder of the computation.

However, when combining region inference and reference tracing garbage collection, a stronger property is needed that guarantees the absence of dangling pointers during evaluation. In systems that use reference tracing garbage collection, the collector may traverse the entire set of values that are reachable via pointers from the root set at any point during evaluation, thus, in such settings, dangling pointers are harmful.

To avoid dangling pointers, it turns out that a sufficient additional requirement on the region typing rules is to force values stored in a closure to live at least as long as the closure itself. Only in special cases does this weakening of the region typing rules alter region annotations. For such an example, consider the following Standard ML program:

```
fun f a = ()
fun g v = fn () => f v
val h = g (2,3)
```

In this program, the function `f` makes no use of its argument. When applied to an argument `v`, the function `g` returns a closure containing `v`, which is a pointer if `v` is boxed. Applying the non-weakened version of region inference to the program yields the following region-annotated program:

```
fun f [] a = ()
fun g [r7] v = (fn () => f[] v)at r7
val h = letregion r8
        in g[r1] (2,3)at r8
        end
```

Notice here that region inference allows functions to take regions as arguments. Due to the region-annotated types that are inferred for `f` and `g`, region inference concludes that the argument pair `(2,3)` passed to `g` can be deallocated after the application of `g`. The result is that, after deallocation of region `r8`, the variable `h` is bound to a closure containing a dangling pointer.

To combine region inference with garbage collection, on the other hand, the region typing rules must ensure that regions holding values captured in a closure live at least as long as the closure itself. Thus, in the combined setting, the result of applying region inference to the binding of `h` yields the following region-annotated version of the binding:

```
val h = g[r1] (2,3)at r1
```

In this region-annotated version of the binding, the pair `(2,3)` is allocated in the global region `r1`, which happens to be the same region in which the closure returned by `g` is allocated.

In short, the contributions of this work are twofold:

- We present experimental data that suggest that, for a range of benchmark programs, the space cost of disallowing dangling pointers is small.
- We present region typing rules that disallow dangling pointers and prove the typing rules to be sound (i.e., that there really are no dangling pointers.)

The lack of dangling pointers, demonstrated by the soundness property, eases the integration of region inference and garbage collection.

1.1 Related Work

We have already mentioned the series of related work on proving type safety results for type systems for region-based memory management [12, 5, 2, 3]. Related to our work in particular is the work by Calcagno, Helsen, and Thiemann [5, 2, 3], who demonstrate that a type safety proof of the region type system based on a store-less contextual semantics can be extended to a store-based contextual semantics, which is suitable also for modeling updatable references. For simplicity, we base our proof of soundness on a store-less contextual semantics.

In [11, page 50], Tofte and Talpin present a side condition to the rule for functions, which they conjecture is sufficient to guarantee the absence of dangling pointers during evaluation. Although the side condition that we present here is inspired by the side condition presented by Tofte and Talpin, several modifications to their side condition was necessary to prove a formal guarantee about the absence of dangling pointers (see Section 3.2). Moreover, the framework developed by Tofte and Talpin is not as suitable for stating a formal guarantee about the absence of dangling pointers as our framework, because their framework is based on a big-step dynamic semantics, which is suitable only for proving properties of terminating programs.

Another line of related work is the work by Morrisett, Felleisen, and Harper on semantics of memory management [8, 7]. Their approach categorizes many different types of memory management techniques, including copying collectors, generational collectors, and type based collectors, in a unified framework based on syntactic type safety proofs.

1.2 Outline

The paper first introduces a region type system, presents a small-step contextual evaluation semantics for the system, and demonstrates, in the style of Helsen and Thiemann [5], a type safety property for the system. Then, in Section 3, a notion of containment is introduced, which defines when all values referred to in an expression are contained in a set of regions. We then present a set of modified typing rules for the region system by adding side conditions to the rules for functions and demonstrate that the type safety guarantee carries over to the modified system. Based on the notion of containment, the paper demonstrates that no dangling pointers are introduced during evaluation. In Section 4, we refine the notion of containment to prove that evaluation does not introduce values in regions that are not present on the region stack, represented by the evaluation context. This consistency property suggests that allocation and deallocation of regions indeed follow a stack discipline.

In Section 5, measurements are presented, which demonstrate that, for a range of benchmark programs, the space costs of avoiding dangling pointers is small.

2. THE REGION TYPE SYSTEM

We first introduce a small region-annotated language with a region type system. The framework that we introduce in this section is much similar to the framework given by Helsen and Thiemann [5], but differs in the way that inaccessibility to values in deallocated regions are modeled. Whereas Helsen and Thiemann "null out" references to deallocated regions (to avoid future access), the framework that we present here keeps track of a set of currently allocated regions and disallows access to regions that are not in this set. For completeness, proofs of important properties of the system are included in Appendix A.

In Section 3, we extend the type system with side conditions that ensure the absence of dangling pointers during evaluation.

2.1 Terms

We assume denumerably infinite sets of *region variables*, ranged over by ρ, and *program variables*, ranged over by x, y, and f. We also assume a denumerably infinite set of integers, ranged over by d. The grammars for *expressions*

(e) and values (v) are defined as follows:

$$
\begin{array}{rcl}
v & ::= & d \; \texttt{in}\; \rho \mid (v_1, v_2)\;\texttt{in}\;\rho \\
 & \mid & \lambda x.e\;\texttt{in}\;\rho \mid \texttt{fix}\; f\;[\vec{\rho}]\; x = e\;\texttt{in}\;\rho \\
e & ::= & v \mid x \mid d\;\texttt{at}\;\rho \mid \texttt{let}\; x = e_1\;\texttt{in}\; e_2 \\
 & \mid & e_1\; e_2 \mid \lambda x.e\;\texttt{at}\;\rho \mid \texttt{letregion}\;\rho\;\texttt{in}\; e \\
 & \mid & \texttt{fix}\; f\;[\vec{\rho}]\; x = e\;\texttt{at}\;\rho \mid e\;[\vec{\rho}]\;\texttt{at}\;\rho \\
 & \mid & (e_1, e_2)\;\texttt{at}\;\rho \mid \texttt{\#1}\; e \mid \texttt{\#2}\; e
\end{array}
$$

All values (i.e., integers, pairs, and closures) are boxed. Thus a value belongs to a particular region, which is denoted by the "in ρ" part of the value. Similarly, all value creating expressions, such as d at ρ and $\lambda x.e$ at ρ, are annotated with information about in which region (i.e., ρ) the value goes at runtime.

In expressions of the forms let x = e_1 in e and $\lambda x.e$ at ρ, the variable x is *bound* in e. In expressions of the form fix f [$\vec{\rho}$] x = e at ρ, the variables f, $\vec{\rho}$, and x are *bound* in e. Similarly for values. In expressions letregion ρ in e, the variable ρ is *bound* in e. As usual, we identify terms up to renaming of bound variables. The *free (program) variables* of some expression (or value) e is written $\mathrm{fpv}(e)$.

Region polymorphism and recursion are supported by the fix construct, which, in combination with the let construct, is sufficient to model the letrec construct supported by the Tofte and Talpin region type system [12].

2.2 Types and Substitutions

We assume denumerably infinite sets of *type variables*, ranged over by α, and *effect variables*, ranged over by ϵ. An *atomic effect*, ranged over by η, is either a region variable or an effect variable. An *arrow effect*, written $\epsilon.\varphi$, is a pair of an effect variable and a set φ of atomic effects.

The grammars for *types* (τ), *type and places* (μ), *type schemes* (σ), and *type scheme and places* (π) are as follows:

$$
\begin{array}{rcl}
\mu & ::= & (\tau, \rho) \\
\tau & ::= & \alpha \mid \texttt{int} \mid \mu_1 \times \mu_2 \mid \mu_1 \xrightarrow{\epsilon.\varphi} \mu_2 \\
\sigma & ::= & \tau \mid \forall \vec{\rho}\vec{\alpha}\vec{\epsilon}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2 \\
\pi & ::= & (\sigma, \rho)
\end{array}
$$

A *type environment* (Γ) is a finite map from program variables to type scheme and places.

For any kind of object o, the *free region variables* and the *free region and effect variables* of o are written $\mathrm{frv}(o)$ and $\mathrm{frev}(o)$, respectively. Moreover, we write $\mathrm{fv}(o)$ to denote the *free type, region, and effect variables* of o. In type schemes of the form $\forall \vec{\rho}\vec{\alpha}\vec{\epsilon}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2$, the variables $\vec{\rho}$, $\vec{\alpha}$, and $\vec{\epsilon}$ are *bound* in $\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2$. We identify type schemes up-to renaming of bound variables. Following Barendregt [1, Appendix C], implicit use of α-conversion eases the formulation and proofs of many of the language properties stated in the following sections.

A *substitution* (S) is a triple $(S^{\mathrm{r}}, S^{\mathrm{t}}, S^{\mathrm{e}})$, where S^{r} is a finite map from region variables to region variables, S^{t} is a finite map from type variables to types, and S^{e} is a finite map from effect variables to arrow effects. The effect of applying a substitution on a particular object is to carry out the three substitutions simultaneously on the three kinds of variables in the object (possibly by renaming of bound variables within the object to avoid capture). For effect sets and arrow effects, substitution is defined as follows [9], assuming $S = (S^{\mathrm{r}}, S^{\mathrm{t}}, S^{\mathrm{e}})$:

$$
\begin{array}{l}
S(\varphi) = \{S^{\mathrm{r}}(\rho) \mid \rho \in \varphi\} \;\cup \\
\qquad \{\eta \mid \exists \epsilon, \epsilon', \varphi'.\epsilon \in \varphi \wedge S^{\mathrm{e}}(\epsilon) = \epsilon'.\varphi' \wedge \eta \in \{\epsilon'\} \cup \varphi'\}
\end{array}
$$

$$S(\epsilon.\varphi) = \epsilon'.(\varphi' \cup S(\varphi)),\ \text{where } S^{\mathrm{e}}(\epsilon) = \epsilon'.\varphi'$$

A type scheme $\sigma = \forall \vec{\rho}\vec{\alpha}\vec{\epsilon}.\tau'$ *generalizes* a type τ *via* $\vec{\rho}\,'$, written $\sigma \geq \tau$ via $\vec{\rho}\,'$, if there exists a substitution $S = (\{\vec{\rho}\,'/\vec{\rho}\}, S^{\mathrm{t}}, S^{\mathrm{e}})$ such that $S(\tau') = \tau$ and $\mathrm{dom}(S^{\mathrm{t}}) = \{\vec{\alpha}\}$, and $\mathrm{dom}(S^{\mathrm{e}}) = \{\vec{\epsilon}\}$.

One can show that generalization is closed under substitution; if $\sigma \geq \tau$ via $\vec{\rho}$, for some σ, τ, and $\vec{\rho}$, and S is a substitution, then $S(\sigma) \geq S(\tau)$ via $S(\vec{\rho})$.

2.3 Typing Rules

The typing rules allow inference of sentences of the form $\Gamma \vdash e : \pi, \varphi$, which says that, in the type environment Γ, the expression (or value) e has type scheme and place π and effect φ. We sometimes write $\vdash e : \pi, \varphi$ to denote the sentence $\{\} \vdash e : \pi, \varphi$.

Values $\boxed{\Gamma \vdash v : \pi, \varphi}$

$$\frac{}{\Gamma \vdash d\;\texttt{in}\;\rho : (\texttt{int}, \rho), \emptyset} \tag{1}$$

$$\frac{\Gamma \vdash v_1 : \mu_1, \emptyset \quad \Gamma \vdash v_2 : \mu_2, \emptyset}{\Gamma \vdash (v_1, v_2)\;\texttt{in}\;\rho : (\mu_1 \times \mu_2, \rho), \emptyset} \tag{2}$$

$$\frac{\Gamma + \{x : \mu_1\} \vdash e : \mu_2, \varphi}{\Gamma \vdash \lambda x.e\;\texttt{in}\;\rho : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \emptyset} \tag{3}$$

$$\frac{\begin{array}{c}\Gamma + \{f : (\forall \vec{\rho}\vec{\epsilon}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), x : \mu_1\} \vdash e : \mu_2, \varphi \\ \mathrm{fv}(\vec{\alpha}\vec{\epsilon}\vec{\rho}) \cap \mathrm{fv}(\Gamma, \varphi) = \emptyset\end{array}}{\Gamma \vdash \texttt{fix}\; f\;[\vec{\rho}]\; x = e\;\texttt{in}\;\rho : (\forall \vec{\rho}\vec{\alpha}\vec{\epsilon}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \emptyset} \tag{4}$$

Expressions $\boxed{\Gamma \vdash e : \pi, \varphi}$

$$\frac{\Gamma \vdash e : \pi, \varphi \quad \varphi' \supseteq \varphi}{\Gamma \vdash e : \pi, \varphi'} \tag{5}$$

$$\frac{}{\Gamma \vdash d\;\texttt{at}\;\rho : (\texttt{int}, \rho), \{\rho\}} \tag{6}$$

$$\frac{\Gamma + \{x : \mu_1\} \vdash e : \mu_2, \varphi}{\Gamma \vdash \lambda x.e\;\texttt{at}\;\rho : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \{\rho\}} \tag{7}$$

$$\frac{\Gamma(x) = \pi}{\Gamma \vdash x : \pi, \emptyset} \tag{8}$$

$$\frac{\Gamma \vdash e : (\sigma, \rho'), \varphi \quad \sigma \geq \tau \text{ via } \vec{\rho}}{\Gamma \vdash e\;[\vec{\rho}]\;\texttt{at}\;\rho : (\tau, \rho), \varphi \cup \{\rho, \rho'\}} \tag{9}$$

$$\frac{\Gamma \vdash e_1 : (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, \rho), \varphi_1 \quad \Gamma \vdash e_2 : \mu', \varphi_2}{\Gamma \vdash e_1\; e_2 : \mu, \varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}} \tag{10}$$

$$\frac{\Gamma \vdash e_1 : \mu_1, \varphi_1 \quad \Gamma \vdash e_2 : \mu_2, \varphi_2}{\Gamma \vdash (e_1, e_2)\ \texttt{at}\ \rho : (\mu_1 \times \mu_2, \rho), \varphi_1 \cup \varphi_2 \cup \{\rho\}} \tag{11}$$

$$\frac{\Gamma \vdash e : (\mu_1 \times \mu_2, \rho), \varphi \quad i \in \{1,2\}}{\Gamma \vdash \texttt{\#}i\ e : \mu_i, \varphi \cup \{\rho\}} \tag{12}$$

$$\frac{\Gamma \vdash e : \mu, \varphi \quad \rho \notin \mathrm{frv}(\Gamma, \mu)}{\Gamma \vdash \texttt{letregion}\ \rho\ \texttt{in}\ e : \mu, \varphi \setminus \{\rho\}} \tag{13}$$

$$\frac{\begin{array}{c}\Gamma + \{f : (\forall \vec{\rho}\vec{\epsilon}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), x : \mu_1\} \vdash e : \mu_2, \varphi \\ \mathrm{fv}(\vec{\alpha}\vec{\epsilon}\vec{\rho}) \cap \mathrm{fv}(\Gamma, \varphi) = \emptyset\end{array}}{\Gamma \vdash \texttt{fix}\ f\ \texttt{[}\vec{\rho}\texttt{]}\ x\ \texttt{=}\ e\ \texttt{at}\ \rho : (\forall \vec{\rho}\vec{\alpha}\vec{\epsilon}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \{\rho\}} \tag{14}$$

$$\frac{\Gamma \vdash e_1 : \pi, \varphi_1 \quad \Gamma + \{x : \pi\} \vdash e_2 : \mu, \varphi_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \mu, \varphi_1 \cup \varphi_2} \tag{15}$$

2.4 Typing Properties

The typing rules possess a series of properties that are important for the type safety and garbage collection safety properties, which we present in later sections.

The following property states that typing judgments are closed under substitution.

PROPOSITION 1 (SUBSTITUTION). *If* $\Gamma \vdash e : \pi, \varphi$ *and* S *is a substitution, then* $S(\Gamma) \vdash S(e) : S(\pi), S(\varphi)$.

PROOF. By induction on the derivation $\Gamma \vdash e : \pi, \varphi$. For completeness, details of the proof appear in Appendix A. □

The following two propositions state that values have no effect and that the typing rules are closed under environment extension.

PROPOSITION 2 (VALUES HAVE NO EFFECT). *If* $\Gamma \vdash e : \pi, \varphi$ *and* e *is a value then* $\Gamma \vdash e : \pi, \emptyset$.

PROOF. By inspection of the typing rules for values. □

PROPOSITION 3 (ENVIRONMENT EXTENSION). *If* $\Gamma \vdash e : \pi, \varphi$ *and* $\mathrm{dom}(\Gamma') \cap \mathrm{dom}(\Gamma) = \emptyset$ *then* $\Gamma + \Gamma' \vdash e : \pi, \varphi$.

PROOF. By induction on the derivation $\Gamma \vdash e : \pi, \varphi$. □

Central to the type preservation argument is the following proposition, which states that the typing rules are appropriately closed under value substitution.

PROPOSITION 4 (VALUE SUBSTITUTION). *If* $\Gamma + \{x : \pi\} \vdash e : \pi', \varphi$ *and* $\vdash v : \pi, \emptyset$ *then* $\Gamma \vdash e[v/x] : \pi', \varphi$.

PROOF. By induction on the derivation $\Gamma + \{x : \pi\} \vdash e : \pi', \varphi$. For completeness, details of the proof appear in Appendix A. □

Finally, the following proposition states that the typing of values dictates the form of the value, canonically.

PROPOSITION 5 (CANONICAL FORMS). *Assume* $\vdash v : (\sigma, \rho), \varphi_0$.

- *If* $\sigma = \texttt{int}$ *then* $v = d\ \texttt{in}\ \rho$*, for some* d
- *If* $\sigma = \mu_1 \times \mu_2$ *then* $v = \texttt{(}v_1, v_2\texttt{)}\ \texttt{in}\ \rho$*, for some* v_1 *and* v_2
- *If* $\sigma = \mu_1 \xrightarrow{\epsilon.\varphi} \mu_2$ *then* $v = \lambda x.e\ \texttt{in}\ \rho$*, for some* x *and* e
- *If* $\sigma = \forall \vec{\rho}\vec{\alpha}\vec{\epsilon}.\tau$ *then* $v = \texttt{fix}\ f\ \texttt{[}\vec{\rho}\texttt{]}\ x\ \texttt{=}\ e\ \texttt{in}\ \rho$*, for some* f*,* x*, and* e

PROOF. By inspection of the typing rules for values, after initial use of (5) in the case φ_0 is non-empty. □

2.5 A Dynamic Semantics

Before we present a small step dynamic semantics for the language, we first introduce the notions of *evaluation contexts* (E) and *instructions* (ι):

$$\begin{array}{lcll}
E_\varphi & ::= & [\cdot] & (\varphi = \emptyset) \\
 & | & \texttt{letregion}\ \rho\ \texttt{in}\ E_{\varphi\setminus\{\rho\}} & (\rho \in \varphi) \\
 & | & E_\varphi\ e \ |\ v\ E_\varphi \ |\ E_\varphi\ \texttt{[}\vec{\rho}\texttt{]}\ \texttt{at}\ \rho & \\
 & | & \texttt{let}\ x\ \texttt{=}\ E_\varphi\ \texttt{in}\ e & \\
 & | & \texttt{(}E_\varphi, e\texttt{)}\ \texttt{at}\ \rho \ |\ \texttt{(}v, E_\varphi\texttt{)}\ \texttt{at}\ \rho \ |\ \texttt{\#}i\ E_\varphi & \\
\iota & ::= & d\ \texttt{at}\ \rho \ |\ \lambda x.e\ \texttt{at}\ \rho \ |\ \texttt{(}v_1, v_2\texttt{)}\ \texttt{at}\ \rho & \\
 & | & \texttt{\#1}\ \texttt{((}v_1, v_2\texttt{)}\ \texttt{in}\ \rho\texttt{)} \ |\ \texttt{\#2}\ \texttt{((}v_1, v_2\texttt{)}\ \texttt{in}\ \rho\texttt{)} & \\
 & | & (\lambda x.e\ \texttt{in}\ \rho)\ v & \\
 & | & (\texttt{fix}\ f\ \texttt{[}\vec{\rho}\texttt{]}\ x\ \texttt{=}\ e\ \texttt{in}\ \rho)\ \texttt{[}\vec{\rho}'\texttt{]}\ \texttt{at}\ \rho &
\end{array}$$

The dynamic semantics that we present is in the style of a contextual dynamic semantics [6]. Notice that contexts E_φ make explicit the set of regions φ bound by `letregion` constructs that encapsulate the hole in the context. The proof of type safety resembles well-known techniques for proving type safety for statically typed languages [6, 13]. The evaluation rules consist of *allocation and deallocation rules*, *reduction rules*, and *a context rule*. The rules are of the form $e \overset{\varphi}{\longmapsto} e'$, which says that, given a set of allocated regions φ, the expression e reduces (in one step) to the expression e'.

Allocation and Deallocation $\boxed{e \overset{\varphi}{\longmapsto} v}$

$$d\ \texttt{at}\ \rho \overset{\varphi\cup\{\rho\}}{\longmapsto} d\ \texttt{in}\ \rho \tag{16}$$

$$\lambda x.e\ \texttt{at}\ \rho \overset{\varphi\cup\{\rho\}}{\longmapsto} \lambda x.e\ \texttt{in}\ \rho \tag{17}$$

$$\texttt{(}v_1, v_2\texttt{)}\ \texttt{at}\ \rho \overset{\varphi\cup\{\rho\}}{\longmapsto} \texttt{(}v_1, v_2\texttt{)}\ \texttt{in}\ \rho \tag{18}$$

$$\texttt{fix}\ f\ \texttt{[}\vec{\rho}\texttt{]}\ x\ \texttt{=}\ e\ \texttt{at}\ \rho \overset{\varphi\cup\{\rho\}}{\longmapsto} \texttt{fix}\ f\ \texttt{[}\vec{\rho}\texttt{]}\ x\ \texttt{=}\ e\ \texttt{in}\ \rho \tag{19}$$

$$\texttt{letregion}\ \rho\ \texttt{in}\ v \overset{\varphi}{\longmapsto} v \tag{20}$$

Reduction $\boxed{e \overset{\varphi}{\longmapsto} e'}$

$$(\lambda x.e\ \texttt{in}\ \rho)\ v \overset{\varphi\cup\{\rho\}}{\longmapsto} e[v/x] \tag{21}$$

$$\texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ e \overset{\varphi}{\longmapsto} e[v/x] \tag{22}$$

$$\begin{array}{c}(\texttt{fix}\ f\ \texttt{[}\vec{\rho}\texttt{]}\ x\ \texttt{=}\ e\ \texttt{in}\ \rho)\texttt{[}\vec{\rho}'\texttt{]}\ \texttt{at}\ \rho' \overset{\varphi\cup\{\rho\}}{\longmapsto} \\ \lambda x.e[\vec{\rho}'/\vec{\rho}][(\texttt{fix}\ f\ \texttt{[}\vec{\rho}\texttt{]}\ x\ \texttt{=}\ e\ \texttt{in}\ \rho)/f]\ \texttt{at}\ \rho'\end{array} \tag{23}$$

$$\texttt{\#1 ((}v_1, v_2\texttt{) in } \rho) \overset{\varphi \cup \{\rho\}}{\longmapsto} v_1 \tag{24}$$

$$\texttt{\#2 ((}v_1, v_2\texttt{) in } \rho) \overset{\varphi \cup \{\rho\}}{\longmapsto} v_2 \tag{25}$$

Context $\boxed{E_\varphi[e] \overset{\varphi'}{\longmapsto} E_\varphi[e']}$

$$\frac{e \overset{\varphi' \cup \varphi}{\longmapsto} e' \quad \varphi \cap \varphi' = \emptyset \quad E_\varphi \neq [\cdot]}{E_\varphi[e] \overset{\varphi'}{\longmapsto} E_\varphi[e']} \tag{26}$$

Notice that evaluation can occur under `letregion` constructs via the context rule. Also notice the rule for region deallocation (20), which expresses the popping of a region ρ from the region stack.

Next, *evaluation* is defined as the least relation formed by the reflexive transitive closure of the relation $\overset{\varphi}{\longmapsto}$, as follows:

Evaluation $\boxed{e \overset{\varphi}{\longmapsto}{}^{*} e'}$

$$\frac{e \overset{\varphi}{\longmapsto} e'}{e \overset{\varphi}{\longmapsto}{}^{*} e'} \tag{27}$$

$$e \overset{\varphi}{\longmapsto}{}^{*} e \tag{28}$$

$$\frac{e_1 \overset{\varphi}{\longmapsto}{}^{*} e_2 \quad e_2 \overset{\varphi}{\longmapsto}{}^{*} e_3}{e_1 \overset{\varphi}{\longmapsto}{}^{*} e_3} \tag{29}$$

We further define $e \Downarrow_\varphi v$ to mean $e \overset{\varphi}{\longmapsto}{}^{*} v$, and $e \Uparrow_\varphi$ to mean that there exists an infinite sequence, $e \overset{\varphi}{\longmapsto} e_1 \overset{\varphi}{\longmapsto} e_2 \overset{\varphi}{\longmapsto} \cdots$.

2.6 Type Safety

To present a general type safety argument, we first state a property saying that a well-typed expression is either a value or can be separated into an evaluation context and a non-value expression:

Proposition 6 (Unique Decomposition). *If* $\vdash e : \pi, \varphi$, *then either*

1. *e is a value; or*
2. *there exists a unique $E_{\varphi'}$, ι, and π' such that $e = E_{\varphi'}[\iota]$ and* $\vdash \iota : \pi', \varphi \cup \varphi'$.

Proof. By induction on the structure of e. For completeness, details of the proof appear in Appendix A. □

A type preservation property (i.e., subject reduction) for the language can then be stated as follows:

Proposition 7 (Type Preservation). *If* $\vdash e : \pi, \varphi$ *and* $e \overset{\varphi}{\longmapsto} e'$ *then* $\vdash e' : \pi, \varphi$.

Proof. By induction on the structure of e, with the use of Proposition 6 and the typing properties stated in Section 2.4. Details of the proof appear in Appendix A. □

Proposition 8 (Progress). *If* $\vdash e : \pi, \varphi$ *then either e is a value or else there exists some e' such that* $e \overset{\varphi}{\longmapsto} e'$.

Proof. By case analysis on the structure of e, using Proposition 6. For completeness, details of the proof are included in Appendix A. □

The progress property implies that well-typed expressions cannot "get stuck" during evaluation, which means that a well-typed program cannot apply a non-function to some argument, project an element from a non-tuple, or access values in regions that are deallocated. In practice, if an implementation accurately models the dynamic semantics, the implementation will not access deallocated memory or "dump core" during evaluation of a program.

Theorem 1 (Type Safety). *If* $\vdash e : \pi, \varphi$, *then either* $e \Uparrow_\varphi$ *or else there exists some v such that* $e \Downarrow_\varphi v$ *and* $\vdash v : \pi, \varphi$.

Proof. By induction on the number of rewriting steps, if $e \overset{\varphi}{\longmapsto}{}^{*} e'$, then by Proposition 7, we have $\vdash e' : \pi, \varphi$. Now, by Proposition 8, either e' is a value or else there exists an e'' such that $e' \overset{\varphi}{\longmapsto} e''$. Thus, either there exists an infinite sequence $e \overset{\varphi}{\longmapsto}{}^{*} e' \overset{\varphi}{\longmapsto} e_1 \overset{\varphi}{\longmapsto} e_2 \overset{\varphi}{\longmapsto} \cdots$, or else $e \Downarrow_\varphi v$ and $\vdash v : \pi, \varphi$. □

3. GARBAGE COLLECTION SAFETY

To guarantee safety of garbage collection, we must ensure that no dangling pointers are introduced during evaluation. The original Tofte-Talpin system [12] does not guarantee the absence of dangling pointers. The solution that we apply here is to add to the typing rules side conditions that guarantee the absence of dangling pointers.

3.1 Containment

First, we define a notion of containment; an expression e is *contained* in a set of regions φ, if the sentence $\varphi \models e$ is derivable from the following rules:

$\boxed{\varphi \models e}$

$$\varphi \models x \tag{30}$$

$$\frac{\rho \in \varphi}{\varphi \models d \texttt{ in } \rho} \tag{31}$$

$$\frac{\rho \in \varphi \quad \varphi \models e}{\varphi \models \lambda x.e \texttt{ in } \rho} \tag{32}$$

$$\frac{\rho \in \varphi \quad \varphi \models v_1 \quad \varphi \models v_2}{\varphi \models \texttt{(}v_1, v_2\texttt{) in } \rho} \tag{33}$$

$$\frac{\rho \in \varphi \quad \varphi \models e}{\varphi \models \texttt{fix } f \texttt{ [}\vec{\rho}\texttt{] } x \texttt{ = } e \texttt{ in } \rho} \tag{34}$$

$$\varphi \models d \texttt{ at } \rho \tag{35}$$

$$\frac{\varphi \models e_1 \quad \varphi \models e_2}{\varphi \models (e_1, e_2)\ \mathtt{at}\ \rho} \tag{36}$$

$$\frac{\varphi \models e}{\varphi \models \lambda x.e\ \mathtt{at}\ \rho} \tag{37}$$

$$\frac{\varphi \models e}{\varphi \models \mathtt{\#}i\ e} \tag{38}$$

$$\frac{\varphi \models e}{\varphi \models \mathtt{fix}\ f\ \mathtt{[}\vec{\rho}\mathtt{]}\ x\ \mathtt{=}\ e\ \mathtt{at}\ \rho} \tag{39}$$

$$\frac{\varphi \models e}{\varphi \models e\ \mathtt{[}\vec{\rho}\mathtt{]}\ \mathtt{at}\ \rho} \tag{40}$$

$$\frac{\varphi \models e_1 \quad \varphi \models e_2}{\varphi \models e_1\ e_2} \tag{41}$$

$$\frac{\varphi \models e_1 \quad \varphi \models e_2}{\varphi \models \mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2} \tag{42}$$

$$\frac{\rho \notin \varphi \quad \varphi \cup \{\rho\} \models e}{\varphi \models \mathtt{letregion}\ \rho\ \mathtt{in}\ e} \tag{43}$$

The containment relation $\varphi \models e$ expresses that for each value v appearing in e, the value v is contained in the set of regions φ and regions bound by `letregion` constructs that encapsulate v in e. Taking the latter kind of regions into account allows for evaluation under `letregion` constructs representing allocated regions on the region stack.

We now state a few properties of the containment relation. The first four properties may be proved by induction on the structure of e; the last two properties may be proved by induction on the structure of $E_{\varphi'}$.

Proposition 9 (Containment Substitution). *If $\varphi \models e$ then $S(\varphi) \models S(e)$.*

Proposition 10 (Containment Extension). *If $\varphi \models e$ and $\varphi' \supseteq \varphi$ then $\varphi' \models e$.*

Proposition 11 (Value Substitution). *If $\varphi \models e$ and $\varphi \models v$ then $\varphi \models e[v/x]$.*

Proposition 12 (Containment Intersection). *If $\varphi \models e$ and $\varphi' \models e$ then $\varphi \cap \varphi' \models e$.*

Proposition 13 (Context Containment). *If $\varphi \models E_{\varphi'}[e]$ then $\varphi \cup \varphi' \models e$.*

Proposition 14 (Context Containment Replace). *If $\varphi \models E_{\varphi'}[e]$ and $\varphi \cup \varphi' \models e'$ then $\varphi \models E_{\varphi'}[e']$.*

3.2 Strengthening of the Region Typing Rules

Before we can state a property saying that, for well-typed programs, dangling pointers are not introduced during evaluation, we introduce a relation $\mathcal{G}$, which we shall use to strengthen the typing rules for functions to avoid dangling pointers during evaluation. The relation is derived from the side condition for functions suggested by Tofte and Talpin in [11, page 50]. With this side condition, the typing rule for functions takes the following form:

$$\frac{\begin{array}{c}\Gamma + \{x : \mu_1\} \vdash e : \mu_2, \varphi \\ \forall y \in \mathrm{fpv}(\lambda x.e\ \mathtt{at}\ \rho).\mathrm{frv}(\Gamma(y)) \subseteq \mathrm{frv}(\varphi)\end{array}}{\Gamma \vdash \lambda x.e\ \mathtt{at}\ \rho : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \{\rho\}} \tag{44}$$

Although this rule appears to be sufficient to guarantee the absence of dangling pointers during evaluation, it suffers from three problems.

First, the side condition is not closed under substitution, because the requirement talks about free region variables, only. This problem breaks the property that typing is closed under substitution (Proposition 1). Modifying the side condition to also include effect variables solves this problem:

$$\forall y \in \mathrm{fpv}(\lambda x.e\ \mathtt{at}\ \rho).\mathrm{frev}(\Gamma(y)) \subseteq \mathrm{frev}(\varphi)$$

With this modified side condition, the typing rules are closed under substitution.

The second problem with the side condition is that it is more restrictive than necessary. It turns out that it is sufficient to require the set of regions appearing in the types of free program variables in the function to be contained in the type scheme and place for the function itself. Thus, the side condition can be refined to read as follows:

$$\forall y \in \mathrm{fpv}(\lambda x.e\ \mathtt{at}\ \rho).\mathrm{frev}(\Gamma(y)) \subseteq \mathrm{frev}(\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho)$$

The third problem with the side condition is that it does not say anything about values substituted for program variables in e. Thus, to prove the property that no dangling pointers are introduced during evaluation, we refine the side condition to require also that values referred to in e are contained in the set of regions present in the type of the function:

$$\begin{array}{l}\forall y \in \mathrm{fpv}(\lambda x.e\ \mathtt{at}\ \rho).\mathrm{frev}(\Gamma(y)) \subseteq \mathrm{frev}(\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho) \\ \text{and } \mathrm{frv}(\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho) \models e\end{array}$$

Intuitively,

1. For each free variable y in the closure, the region and effect variables needed to hold y are contained in the region and effect variables occuring free in the type and place of the function.

2. All values in the closure body are contained in the region variables occuring free in the type and place of the function.

To ease the notation, we define a relation $\mathcal{G}$, which is parameterized over an environment Γ, a function body e, a set of function parameters X, and the type scheme and place π of the function:

Definition 1 (GC Safety).

$$\begin{array}{rcl}\mathcal{G}(\Gamma, e, X, \pi) & = & \forall y \in \mathrm{fpv}(e) \setminus X\,.\,\mathrm{frev}(\Gamma(y)) \subseteq \mathrm{frev}(\pi) \\ & & \text{and } \mathrm{frv}(\pi) \models e\end{array}$$

The modified type system, which we shall show is safe for garbage collection, is the system in Section 2.3, with rules (3), (4), (7), and (14) modified to include an additional side condition as follows:

$$\frac{\begin{array}{c}\Gamma + \{x : \mu_1\} \vdash e : \mu_2, \varphi \\ \mu = (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho) \quad \mathcal{G}(\Gamma, e, \{x\}, \mu)\end{array}}{\Gamma \vdash \lambda x.e \texttt{ in } \rho : \mu, \emptyset} \tag{45}$$

$$\frac{\begin{array}{c}\Gamma + \{f : (\forall \vec{\rho}\vec{\epsilon}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), x : \mu_1\} \vdash e : \mu_2, \varphi \\ \mathrm{fv}(\vec{\alpha}\vec{\epsilon}\vec{\rho}) \cap \mathrm{fv}(\Gamma, \varphi) = \emptyset \\ \pi = (\forall \vec{\rho}\vec{\alpha}\vec{\epsilon}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho) \quad \mathcal{G}(\Gamma, e, \{f, x\}, \pi)\end{array}}{\Gamma \vdash \texttt{fix } f \texttt{ [}\vec{\rho}\texttt{] } x \texttt{ = } e \texttt{ in } \rho : \pi, \emptyset} \tag{46}$$

$$\frac{\begin{array}{c}\Gamma + \{x : \mu_1\} \vdash e : \mu_2, \varphi \\ \mu = (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho) \quad \mathcal{G}(\Gamma, e, \{x\}, \mu)\end{array}}{\Gamma \vdash \lambda x.e \texttt{ at } \rho : \mu, \{\rho\}} \tag{47}$$

$$\frac{\begin{array}{c}\Gamma + \{f : (\forall \vec{\rho}\vec{\epsilon}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), x : \mu_1\} \vdash e : \mu_2, \varphi \\ \mathrm{fv}(\vec{\alpha}\vec{\epsilon}\vec{\rho}) \cap \mathrm{fv}(\Gamma, \varphi) = \emptyset \\ \pi = (\forall \vec{\rho}\vec{\alpha}\vec{\epsilon}.\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho) \quad \mathcal{G}(\Gamma, e, \{f, x\}, \pi)\end{array}}{\Gamma \vdash \texttt{fix } f \texttt{ [}\vec{\rho}\texttt{] } x \texttt{ = } e \texttt{ at } \rho : \pi, \{\rho\}} \tag{48}$$

3.3 Type Safety for the Modified System

Our first task is to carry over the type safety result (Theorem 1) to the modified system. To do so, we must verify that all the propositions that we have demonstrated for the system in Section 2.3 also hold for the modified system.

As mentioned earlier, because the garbage collection condition that we have added to the rules is closed under substitution, it is easy to verify that Proposition 1 also holds for the modified system. Moreover, Proposition 2, Proposition 3, and Proposition 5 are easily shown for the modified system. To demonstrate the value substitution property (Proposition 4), we first demonstrate two auxiliary properties, stating (1) that no part of a well-typed value is contained in a region that does not appear in the type scheme and place of the value and (2) that the relation $\mathcal{G}$ is closed under value substitution.

PROPOSITION 15 (VALUE CLOSEDNESS). *If* $\vdash v : \pi, \emptyset$ *then* $\mathrm{frv}(\pi) \models v$.

PROOF. By induction on the typing rules for values. The interesting case is the case for functions:

CASE $v = \lambda x.e \texttt{ in } \rho$. From (45) and Definition 1, we have $\mathrm{frv}(\pi) \models e$ and $\rho \in \mathrm{frv}(\pi)$. From (32), we can conclude $\mathrm{frv}(\pi) \models v$. □

PROPOSITION 16 (GC SAFETY VALUE SUBSTITUTION). *If* $\mathcal{G}(\Gamma + \{x : \pi\}, e, X, \pi')$ *and* $\vdash v : \pi, \emptyset$ *and* $x \notin X$ *then* $\mathcal{G}(\Gamma, e[v/x], X, \pi')$.

PROOF. If $x \notin \mathrm{fpv}(e)$ then $\mathcal{G}(\Gamma, e[v/x], X, \pi')$ follows trivially. Now, assume $x \in \mathrm{fpv}(e)$. By Definition 1, we have

$$\begin{array}{l}\forall y \in \mathrm{fpv}(e) \setminus X \,.\, \mathrm{frev}((\Gamma + \{x : \pi\})(y)) \subseteq \mathrm{frev}(\pi') \\ \text{and } \mathrm{frv}(\pi') \models e\end{array} \tag{49}$$

It follows from $\vdash v : \pi, \emptyset$ and Proposition 15 that we have $\mathrm{frv}(\pi) \models v$. Thus, because $x \in \mathrm{fpv}(e)$ and (49) imply $\mathrm{frv}(\pi) \subseteq \mathrm{frv}(\pi')$, we have from Proposition 10 that $\mathrm{frv}(\pi') \models v$. It now follows from (49) and Proposition 11 that we have

$$\mathrm{frv}(\pi') \models e[v/x] \tag{50}$$

Because $\vdash v : \pi, \emptyset$ implies $\mathrm{fpv}(v) = \emptyset$, we can derive from (49) that

$$\forall y \in \mathrm{fpv}(e[v/x]) \setminus X \,.\, \mathrm{frev}(\Gamma(y)) \subseteq \mathrm{frev}(\pi') \tag{51}$$

Now, from Definition 1 and from (50) and (51), we can conclude $\mathcal{G}(\Gamma, e[v/x], X, \pi')$, as required. □

Using Proposition 16, it is straightforward to verify that Proposition 6, Proposition 7, and Proposition 8 (and thus Theorem 1) carry over to the modified system.

3.4 Absence of Dangling Pointers

Recall that the relation $\varphi \models e$ expresses that each value v within e is contained in the set of regions φ and regions bound by `letregion` constructs in e encapsulating v. Thus, the relation expresses that if each region within the set φ represents an allocated region, then no dangling pointer appears in e.

The following theorem states that, for well-typed programs, dangling pointers are not introduced by an evaluation step:

THEOREM 2 (NO DANGLING POINTERS). *If* $\vdash e : \pi, \varphi$ *and* $\varphi' \models e$ *and* $e \overset{\varphi'}{\longmapsto} e'$ *then* $\varphi' \models e'$.

PROOF. By induction on the structure of e. We proceed by case analysis on the derivation $e \overset{\varphi'}{\longmapsto} e'$.

CASE $e = \texttt{\#1 (}v_1, v_2\texttt{) in } \rho$. We have $e' = v_1$. From (38) and (33), we have $\varphi' \models v_1$, as required.

CASE $e = (\lambda x.e_0 \texttt{ in } \rho)\ v$. We have $e' = e_0[v/x]$. From (41) and (32), we have $\varphi' \models e_0$ and $\varphi' \models v$. By applying Proposition 11, we have $\varphi' \models e'$, as required.

CASE $e = \texttt{letregion } \rho \texttt{ in } v$. We have $e' = v$. From assumptions, (13), and Proposition 2, we have $\vdash v : \pi, \emptyset$ and $\rho \notin \mathrm{frv}(\pi)$. From Proposition 15, we have $\mathrm{frv}(\pi) \models v$. Moreover, from (43) and assumptions, we have $\varphi' \cup \{\rho\} \models v$, thus, we can apply Proposition 12 to get $\mathrm{frv}(\pi) \cap (\varphi' \cup \{\rho\}) \models v$. Because $\rho \notin \mathrm{frv}(\pi)$, we have $\mathrm{frv}(\pi) \cap \varphi' \models v$. From Proposition 10, we can now conclude $\varphi' \models v$, as required.

CASE $e = E_{\varphi''}[e_1]$, where $E_{\varphi''} \neq [\cdot]$. We have $e' = E_{\varphi''}[e_2]$ and $e_1 \overset{\varphi' \cup \varphi''}{\longmapsto} e_2$ and $\varphi' \cap \varphi'' = \emptyset$. From Proposition 6, we have there exists π' such that $\vdash e_1 : \pi', \varphi' \cup \varphi''$. From assumptions and Proposition 13, we have $\varphi' \cup \varphi'' \models e_1$. We can now apply the induction hypothesis to get $\varphi' \cup \varphi'' \models e_2$. Now, from Proposition 14, we have $\varphi' \models e'$, as required. □

We define an expression to be *value-free* if it contains no values. When combined with properties of type preservation (Proposition 7) and progress (Proposition 8), Theorem 2 guarantees that no dangling pointers are introduced during evaluation of well-typed value-free programs.

4. REGION CONSISTENCY

In this section, we refine the containment relation to prove that evaluation does not introduce values in regions that are not present on the region stack, represented by the evaluation context.

In the syntactic approaches to proving type soundness for the region calculus [5, 2, 3], it is possible to write programs that type check, but for which regions may not be allocated in a stack-like manner. This problem also appears in the region calculus presented in the previous sections. Consider the following program:

$$\begin{array}{rcl} e & \equiv & (\ \texttt{letregion}\ \rho\ \texttt{in} \\ & & \quad\ \ \texttt{#1(}3\ \texttt{in}\ \rho_0, 4\ \texttt{in}\ \rho\texttt{)}\ \texttt{in}\ \rho, \\ & & \quad \texttt{letregion}\ \rho'\ \texttt{in} \\ & & \quad\ \ \texttt{#1(}5\ \texttt{in}\ \rho_0, 6\ \texttt{in}\ \rho'\texttt{)}\ \texttt{in}\ \rho' \\ & &)\ \texttt{at}\ \rho_0 \end{array} \tag{52}$$

With the use of the typing rules in Section 2.3, we can conclude $\vdash e : ((\texttt{int}, \rho_0) \times (\texttt{int}, \rho_0), \rho_0), \{\rho_0\}$. Moreover, the evaluation rules in Section 2.5 allow us to conclude that e evaluates to the value $\texttt{(}3\ \texttt{in}\ \rho_0, 5\ \texttt{in}\ \rho_0\texttt{)}\ \texttt{in}\ \rho_0$ in four steps. If regions were to be allocated in a stack-like manner, only one of the regions ρ and ρ' could be allocated at the same time, which contradicts the fact that e contains values in both ρ and ρ'. (The example carries over to the store-based small-step operational semantics considered in [3].)

By refining the notion of containment, Theorem 2 can be refined to disallow allocation in regions that are neither global nor present on the region stack, represented by the evaluation context. Thus, starting with a well-typed value-free program, evaluation introduces no dangling pointers and allows for regions to be allocated and deallocated in a stack-like manner.

4.1 Refinement of Containment

We refine the notion of containment to include two different relations $\varphi \models_{\mathrm{v}} e$ and $\varphi \models_{\mathrm{c}} e$ of which the former expresses that all values in e are contained in regions in φ. The latter relation expresses that when e can be written on the form $E_{\varphi'}[e']$, values in e' may be contained in regions in the set $\varphi \cup \varphi'$, where φ' are regions on the stack represented by the evaluation context $E_{\varphi'}$.

The relation $\varphi \models_{\mathrm{v}} e$ is defined similarly to the relation $\varphi \models e$ from Section 3.1, except that the rule for expressions of the form $\texttt{letregion}\ \rho\ \texttt{in}\ e'$ does not add ρ to the set of allocated regions when considering e':

$$\boxed{\varphi \models_{\mathrm{v}} e}$$

$$\vdots$$

rules (30) to (42) with $\models_{\mathrm{v}}$ substituted for $\models$

$$\vdots$$

$$\frac{\rho \notin \varphi \quad \varphi \models_{\mathrm{v}} e}{\varphi \models_{\mathrm{v}} \texttt{letregion}\ \rho\ \texttt{in}\ e} \tag{53}$$

The definition of the relation $\varphi \models_{\mathrm{c}} e$ closely follows the definition of evaluation contexts:

$$\boxed{\varphi \models_{\mathrm{c}} e}$$

$$\varphi \models_{\mathrm{c}} x \tag{54}$$

$$\frac{\varphi \models_{\mathrm{v}} v}{\varphi \models_{\mathrm{c}} v} \tag{55}$$

$$\frac{\rho \notin \varphi \quad \varphi \cup \{\rho\} \models_{\mathrm{c}} e}{\varphi \models_{\mathrm{c}} \texttt{letregion}\ \rho\ \texttt{in}\ e} \tag{56}$$

$$\frac{\varphi \models_{\mathrm{c}} e \quad \varphi \models_{\mathrm{v}} e'}{\varphi \models_{\mathrm{c}} e\ e'} \tag{57}$$

$$\frac{\varphi \models_{\mathrm{v}} v \quad \varphi \models_{\mathrm{c}} e}{\varphi \models_{\mathrm{c}} v\ e} \tag{58}$$

$$\frac{\varphi \models_{\mathrm{c}} e}{\varphi \models_{\mathrm{c}} e\ \texttt{[}\vec{\rho}\texttt{]}\ \texttt{at}\ \rho} \tag{59}$$

$$\frac{\varphi \models_{\mathrm{c}} e \quad \varphi \models_{\mathrm{v}} e'}{\varphi \models_{\mathrm{c}} \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e'} \tag{60}$$

$$\frac{\varphi \models_{\mathrm{c}} e \quad \varphi \models_{\mathrm{v}} e'}{\varphi \models_{\mathrm{c}} \texttt{(}e, e'\texttt{)}\ \texttt{at}\ \rho} \tag{61}$$

$$\frac{\varphi \models_{\mathrm{v}} v \quad \varphi \models_{\mathrm{c}} e}{\varphi \models_{\mathrm{c}} \texttt{(}v, e\texttt{)}\ \texttt{at}\ \rho} \tag{62}$$

$$\frac{\varphi \models_{\mathrm{c}} e}{\varphi \models_{\mathrm{c}} \texttt{#}i\ e} \tag{63}$$

$$\varphi \models_{\mathrm{c}} d\ \texttt{at}\ \rho \tag{64}$$

$$\frac{\varphi \models_{\mathrm{v}} e}{\varphi \models_{\mathrm{c}} \lambda x.e\ \texttt{at}\ \rho} \tag{65}$$

$$\frac{\varphi \models_{\mathrm{v}} e}{\varphi \models_{\mathrm{c}} \texttt{fix}\ f\ \texttt{[}\vec{\rho}\texttt{]}\ x\ \texttt{=}\ e\ \texttt{at}\ \rho} \tag{66}$$

The following property relates the two containment relations:

Proposition 17 (Containment Weakening).
If $\varphi \models_{\mathrm{v}} e$ then $\varphi \models_{\mathrm{c}} e$.

Proof By induction on the structure of e. □

Notice that, for the expression e defined by equation (52), there exists no φ such that $\varphi \models_{\mathrm{c}} e$.

The properties presented in Section 3.1 carries over to the refined definition of containment. Again, the first four properties may be proved by induction on the structure of e, whereas the last two properties may be proved by induction on the structure of $E_{\varphi'}$.

Proposition 18 (Containment Substitution).
If $\varphi \models_{\mathrm{v}} e$ then $S(\varphi) \models_{\mathrm{v}} S(e)$. If $\varphi \models_{\mathrm{c}} e$ then $S(\varphi) \models_{\mathrm{c}} S(e)$.

Proposition 19 (Containment Extension).
If $\varphi \models_{\mathrm{v}} e$ and $\varphi' \supseteq \varphi$ then $\varphi' \models_{\mathrm{v}} e$.
If $\varphi \models_{\mathrm{c}} e$ and $\varphi' \supseteq \varphi$ then $\varphi' \models_{\mathrm{c}} e$.

Proposition 20 (Value Substitution).
If $\varphi \models_{\mathrm{v}} e$ and $\varphi \models_{\mathrm{v}} v$ then $\varphi \models_{\mathrm{v}} e[v/x]$.
If $\varphi \models_{\mathrm{c}} e$ and $\varphi \models_{\mathrm{v}} v$ then $\varphi \models_{\mathrm{c}} e[v/x]$.

Proposition 21 (Containment Intersection).
If $\varphi \models_v e$ *and* $\varphi' \models_v e$ *then* $\varphi \cap \varphi' \models_v e$.
If $\varphi \models_c e$ *and* $\varphi' \models_c e$ *then* $\varphi \cap \varphi' \models_c e$.

Proposition 22 (Context Containment).
If $\varphi \models_c E_{\varphi'}[e]$ *then* $\varphi \cup \varphi' \models_c e$.

Proposition 23 (Context Containment Replace).
If $\varphi \models_c E_{\varphi'}[e]$ *and* $\varphi \cup \varphi' \models_c e'$ *then* $\varphi \models_c E_{\varphi'}[e']$.

The definition of the garbage collection safety relation $\mathcal{G}$ (Definition 1) is refined as follows to accommodate for the refinement of containment:

Definition 2 (GC Safety — Refined).

$$\begin{aligned}\mathcal{G}(\Gamma, e, X, \pi) \quad = \quad & \forall y \in \mathrm{fpv}(e) \setminus X.\mathrm{frev}(\Gamma(y)) \subseteq \mathrm{frev}(\pi) \\ & \text{and } \mathrm{frv}(\pi) \models_v e\end{aligned}$$

For the refined system, no other changes are made to the typing rules and no changes are made to the evaluation rules. Again, because the garbage collection safety relation is closed under substitution, Proposition 1 carries over to the refined system, as well as Proposition 2, Proposition 3, and Proposition 5.

The value closedness property (Proposition 15) is refined as follows:

Proposition 24 (Value Closedness — Refined).
If $\vdash v : \pi, \emptyset$ *then* $\mathrm{frv}(\pi) \models_v v$.

Proof. By induction on the typing rules for values. The proof is similar to the proof for Proposition 15. □

From Proposition 24 and the refined definition of $\mathcal{G}$, it is straightforward to demonstrate that type preservation (Proposition 7), progress (Proposition 8), and type safety (Theorem 1) carry over to the refined system.

The following refined version of Theorem 2 states that, for well-typed programs, dangling pointers are not introduced by an evaluation step and no value in the program is allocated in a region that is not present on the region stack, represented by the evaluation context.

Theorem 3 (No Dangling Pointers — Refined).
If $\vdash e : \pi, \varphi$ *and* $\varphi' \models_c e$ *and* $e \overset{\varphi'}{\longmapsto} e'$ *then* $\varphi' \models_c e'$.

Proof. By induction on the structure of e. We proceed by case analysis on the derivation $e \overset{\varphi'}{\longmapsto} e'$.

Case $e =$ `#1` (v_1, v_2) `in` ρ. We have $e' = v_1$. From assumptions and from (63) and (62), we have $\varphi' \models_v v_1$. From (55), we can conclude $\varphi' \models_c v_1$, as required.

Case $e = (\lambda x.e_0$ `in` $\rho)\ v$. We have $e' = e_0[v/x]$. From assumptions and from (58) and (55) and from the refined version of (32), we have $\varphi' \models_v e_0$ and $\varphi' \models_v v$. By applying Proposition 20, we have $\varphi' \models_v e'$. We can now apply Proposition 17 to get $\varphi' \models_c e'$, as required.

Case $e =$ `letregion` ρ `in` v. We have $e' = v$. From assumptions, (13), and Proposition 2, we have $\vdash v : \pi, \emptyset$ and $\rho \notin \mathrm{frv}(\pi)$. From Proposition 24, we have $\mathrm{frv}(\pi) \models_v v$. Moreover, from (56) and assumptions, we have $\varphi' \cup \{\rho\} \models_c v$. Because v is a value, (55) must have been applied, thus, we have $\varphi' \cup \{\rho\} \models_v v$. We can now apply Proposition 21 to get $\mathrm{frv}(\pi) \cap (\varphi' \cup \{\rho\}) \models_v v$. Because $\rho \notin \mathrm{frv}(\pi)$, we have $\mathrm{frv}(\pi) \cap \varphi' \models_v v$. From Proposition 19 and from (55), we can now conclude $\varphi' \models_c v$, as required.

Case $e = E_{\varphi''}[e_1]$, where $E_{\varphi''} \neq [\cdot]$. We have $e' = E_{\varphi''}[e_2]$ and $e_1 \overset{\varphi' \cup \varphi''}{\longmapsto} e_2$ and $\varphi' \cap \varphi'' = \emptyset$. From Proposition 6, we have there exists π' such that $\vdash e_1 : \pi', \varphi' \cup \varphi''$. From assumptions and Proposition 22, we have $\varphi' \cup \varphi'' \models_c e_1$. We can now apply the induction hypothesis to get $\varphi' \cup \varphi'' \models_c e_2$. Now, from Proposition 23, we have $\varphi' \models_c e'$, as required. □

When combined with the properties of type preservation (Proposition 7) and progress (Proposition 8), Theorem 3 guarantees that no dangling pointers are introduced during evaluation of well-typed value-free programs and that evaluation does not introduce values in regions that are not present on the region stack, represented by the evaluation context. As a consequence, allocation and deallocation of regions indeed follow a stack discipline.

5. MEASUREMENTS

In this section we present measurements demonstrating that the strengthened typing rules for functions, as presented in Section 3.2, do not change the result of region inference dramatically. The memory discipline is implemented for all of Standard ML in the ML Kit compiler [10]. We present both the number of static region annotation changes to the generated program caused by the strengthening and the difference in memory usage between programs compiled with and without the strengthened typing rules. When the strengthened rules are used, we also present the memory usage of the program when region-based memory management is combined with a copying reference tracing garbage collector [4].

The benchmark programs are presented in Table 1 and span from small micro-benchmarks (`fib35`, `tak`, and `msort`) to larger programs, such as `vliw` and `mlyacc`, that solve real-world problems. The `Lines` column shows the size of each benchmark. None of the benchmark programs, except `msort`, `kitlife`, and `kitkb`, has been optimized for region inference. The benchmark programs `fib35` and `tak` use only the runtime stack for allocation. Benchmark statistics for

Table 1: The benchmark programs.

Program	Lines	Description
`vliw`	3676	VLIW instruction scheduler
`logic`	346	SML/NJ benchmark program
`zebra`	302	Solves the Zebra puzzle
`tyan`	1018	Gröbner Basis calculation
`tsp`	493	Traveling salesman problem
`mpuz`	142	Emacs M-x mpuz puzzle
`DLX`	2836	DLX RISC instruction simulation
`ratio`	619	Image analysis
`lexgen`	1318	Lexer generation
`mlyacc`	7353	Parser generation
`simple`	1052	Spherical fluid-dynamics program
`professor`	276	Solves puzzle by exhaustive search
`fib35`	9	The Fibbonachi micro-benchmark
`tak`	17	The Tak micro-benchmark
`msort`	81	Sorting 100,000 integers
`kitlife`	230	The game of life
`kitkb`	725	Knuth-Bendix completion

Table 2: Benchmark statistics.

Program	Memory usage (bytes)			Number of static differences
	dangling pointers	no dangling pointers		
	NO GC	NO GC	GC	
`vliw`	4320k	4208k	**2608k**	143
`logic`	128M	128M	**812k**	14
`zebra`	6728k	6660k	**620k**	16
`tyan`	199M	197M	**2356k**	26
`tsp`	**3456k**	**3456k**	6316k	0
`mpuz`	**480k**	**480k**	520k	0
`DLX`	**2916k**	**2916k**	3016k	4
`ratio`	2736k	2736k	**1456k**	1
`lexgen`	19M	19M	**3496k**	19
`mlyacc`	4552k	4644k	**2860k**	93
`simple`	**1204k**	1208k	1836k	120
`professor`	4624k	4592k	**564k**	5
`fib35`	**436k**	**436k**	452k	0
`tak`	**436k**	**436k**	452k	0
`msort`	**3436k**	**3436k**	4608k	2
`kitlife`	**468k**	**468k**	528k	0
`kitkb`	**1072k**	**1072k**	1188k	26

the different programs are presented in Table 2. The benchmark programs are run on a 750Mhz Pentium III Linux box with 512Mb RAM. Memory usage (resident set size) is measured in kilobytes using the `/proc` special file-system under the Linux operating system. The experiments are performed with the ML Kit version 4.1.3. The first two columns show memory usage (resident set size) for the different benchmark programs with and without dangling pointers. For both columns, garbage collection and tagging of values are disabled. The third column shows memory usage when no dangling pointers are allowed and garbage collection is enabled. The fourth column shows the number of static region annotation changes to the program enforced by the strengthened typing rules.

The numbers demonstrate the following properties:

- Reference tracing garbage collection improves memory usage for most of those programs that are not optimized for region inference (e.g., `logic` and `lexgen`).
- When reference tracing garbage collection is disabled, there are only two programs out of the 17 benchmarks (i.e., `mlyacc` and `simple`) for which the enforced absence of dangling pointers increases memory usage.
- Due to the necessity of tagging values, such as reals, when garbage collection is enabled, memory usage is higher for some programs when garbage collection is enabled than when disabled; this behavior is demonstrated by the `tsp` benchmark.

6. CONCLUSION AND FUTURE WORK

This paper demonstrates the safety of combining region inference and reference tracing garbage collection by refining the region typing rules to guarantee the absence of dangling pointers during execution of a program.

The paper also demonstrates, experimentally, that the strengthened region type system has little influence on memory usage of compiled programs.

There are several directions for future work. First, there are aspects of the integration of region inference and garbage collection in the ML Kit that can be improved. In particular, arranging that garbage collection can be initiated at arbitrary allocation points—instead of only at function entry points—may improve memory usage for some programs. Also, current research investigates the possibility of combining region inference with a tag-free garbage collection scheme.

7. ACKNOWLEDGMENTS

Thanks to Peter Sestoft and Lars Birkedal for their careful reading of a draft of this paper and to Niels Hallenberg for his work on the garbage collector in the ML Kit.

8. REFERENCES

[1] Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Elsevier Science Publishers, 1984. Volume 103 of Studies in Logic and the Foundations of Mathematics, North-Holland, Amsterdam, second edition.

[2] Cristiano Calcagno. Stratified operational semantics for safety and correctness of the region calculus. In *Procedings of ACM Symposium on Principles of Programming Languages (POPL'01).* ACM Press, January 2001.

[3] Cristiano Calcagno, Simon Helsen, and Peter Thiemann. Syntactic type soundness results for the region calculus. *Information and Computation*, 173(2), 2002.

[4] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02).* ACM Press, June 2002. Berlin, Germany.

[5] Simon Helsen and Peter Thiemann. Syntactic type soundness for the region calculus. In *Proceedings of the 4th International Workshop on Higher Order Operational Techniques in Semantics*, September 2000. Published in Volume 41(3) of the Electronic Notes in Theoretical Computer Science.

[6] Greg Morrisett. *Compiling with Types.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, December 1995.

[7] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Procedings of the International Conference on Functional Programming Languages and Computer Architecture*, pages 66–77, June 1995. San Diego.

[8] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. Technical Report CMU-CS-96-176, Carnegie Mellon University, September 1996.

[9] Mads Tofte and Lars Birkedal. Unification and polymorphism in region inference. *Proof, Language, and Interaction. Essays in Honour of Robin Milner*, May 2000. (25 pages).

[10] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions in the ML Kit (for version 4). Technical Report TR-2001-07, IT University of Copenhagen, October 2001.

[11] Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages. Technical Report DIKU-report 93/15, Department of Computer Science, University of Copenhagen, 1993.

[12] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

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

APPENDIX

A. PROOFS

Proposition 1 (Substitution) *If $\Gamma \vdash e : \pi, \varphi$ and S is a substitution, then $S(\Gamma) \vdash S(e) : S(\pi), S(\varphi)$.*

Proof. By induction on the derivation $\Gamma \vdash e : \pi, \varphi$. The cases for integers, pairs, projections, and identifiers are trivial.

Case $e = \lambda x.e'$ `in` ρ. From (3), we have $\Gamma \vdash e : (\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2, \rho), \emptyset$ and $\Gamma + \{x : \mu_1\} \vdash e' : \mu_2, \varphi$. By induction, we have $S(\Gamma) + \{x : S(\mu_1)\} \vdash S(e') : S(\mu_2), S(\varphi)$. Let $S(\epsilon) = \epsilon'.\varphi'$. From the definition of substitution, we have $S(\epsilon.\varphi) = \epsilon'.(\varphi' \cup S(\varphi))$. Now, by applying (5), we have $S(\Gamma) + \{x : S(\mu_1)\} \vdash S(e') : S(\mu_2), S(\varphi) \cup \varphi'$. We can now apply (3) to get $S(\Gamma) \vdash S(e) : S(\mu_1 \xrightarrow{\epsilon.\varphi} \mu_2), \emptyset$, as required.

Case $e = e_1\ e_2$. From (10), we have $\Gamma \vdash e : \mu, \varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}$ and $\Gamma \vdash e_1 : (\mu' \xrightarrow{\epsilon.\varphi_0} \mu, \rho), \varphi_1$ and $\Gamma \vdash e_2 : \mu', \varphi_2$. By induction we have $S(\Gamma) \vdash S(e_1) : (S(\mu') \xrightarrow{S(\epsilon.\varphi_0)} S(\mu), S(\rho)), S(\varphi_1)$ and $S(\Gamma) \vdash S(e_2) : S(\mu'), S(\varphi_2)$. Let $S(\epsilon) = \epsilon'.\varphi'$. It follows from the definition of substitution that $S(\epsilon.\varphi_0) = \epsilon'.(\varphi' \cup S(\varphi_0))$. We can now apply (10) to get $S(\Gamma) \vdash S(e) : S(\mu), S(\varphi_0) \cup \varphi' \cup S(\varphi_1) \cup S(\varphi_2) \cup \{\epsilon', S(\rho)\}$. From the definition of substitution, it follows that $S(\{\epsilon, \rho\}) = \{\epsilon', S(\rho)\} \cup \varphi'$, thus, we have $S(\Gamma) \vdash S(e) : S(\mu), S(\varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\})$, as required.

Case $e =$ `letregion` ρ `in` e'. From (13), we have $\Gamma \vdash e : \mu, \varphi \setminus \{\rho\}$ and $\Gamma \vdash e' : \mu, \varphi$ and $\rho \notin \text{frev}(\Gamma, \mu)$. By induction, we have $S(\Gamma) \vdash S(e') : S(\mu), S(\varphi)$. By renaming of bound names, we can assume $\rho \notin \text{frev}(S(\Gamma), S(\mu))$, thus, we can apply (13), to get $S(\Gamma) \vdash$ `letregion` ρ `in` $S(e') : S(\mu), S(\varphi) \setminus \{\rho\}$. Also by renaming of bound names, we can assume `letregion` ρ `in` $S(e') = S($`letregion` ρ `in` $e')$ and $S(\varphi) \setminus \{\rho\} = S(\varphi \setminus \{\rho\})$, thus, we have $S(\Gamma) \vdash S(e) : S(\mu), S(\varphi \setminus \{\rho\})$, as required.

Case $e = e'$ `[`$\vec{\rho}$`]` `at` ρ. From (9), we have $\Gamma \vdash e' : (\sigma, \rho'), \varphi$ and $\sigma \geq \tau$ via $\vec{\rho}$. By induction, we have $S(\Gamma) \vdash S(e') : (S(\sigma), S(\rho')), S(\varphi)$. Because generalization is closed under substitution, we have $S(\sigma) \geq S(\tau)$ via $S(\vec{\rho})$, thus, from (9), we can conclude $S(\Gamma) \vdash S(e) : S(\tau, \rho), S(\varphi \cup \{\rho, \rho'\})$, as required.

Case Rule (5). We have $\Gamma \vdash e : \pi, \varphi$ and $\Gamma \vdash e : \pi, \varphi'$ and $\varphi' \supseteq \varphi$. By induction. we have $S(\Gamma) \vdash S(e) : S(\mu), S(\varphi)$. From the definition of substitution, it follows that $\varphi' \supseteq \varphi$ implies $S(\varphi') \supseteq S(\varphi)$, thus, we can apply (5) to get $S(\Gamma) \vdash S(e) : S(\mu), S(\varphi')$, as required. □

Proposition 4 (Value Substitution) *If $\Gamma + \{x : \pi\} \vdash e : \pi', \varphi$ and $\vdash v : \pi, \emptyset$ then $\Gamma \vdash e[v/x] : \pi', \varphi$.*

Proof. By induction on the derivation $\Gamma + \{x : \pi\} \vdash e : \pi', \varphi$.

Case $e = y$. From assumptions and (8), we have $\Gamma + \{x : \pi\} \vdash y : \pi', \varphi$ and $(\Gamma + \{x : \pi\})(y) = \pi'$ and $\varphi = \emptyset$. If $y \neq x$, we have $e[v/x] = y$, thus, because $\Gamma(y) = \pi'$, we can conclude from (8) that $\Gamma \vdash e[v/x] : \pi', \varphi$, as required. Otherwise, $y = x$, thus $e[v/x] = v$ and $\pi = \pi'$. From assumptions and Proposition 3, we have $\Gamma \vdash e[v/x] : \pi', \varphi$, as required.

Case $e = \lambda y.e'$ `at` ρ. From assumptions and (7), we have $\Gamma + \{x : \pi, y : \mu\} \vdash e' : \mu', \varphi'$ and $\varphi = \{\rho\}$ and $\pi' = (\mu \xrightarrow{\epsilon.\varphi'} \mu', \rho)$. By renaming of bound variables, we can assume $x \neq y$, thus, we can apply the induction hypothesis to get $\Gamma + \{y : \mu\} \vdash e'[v/x] : \mu', \varphi'$. By applying (7), we have $\Gamma \vdash \lambda y.e'[v/x] : \pi', \varphi$, as required.

The remaining cases follow similarly. □

Proposition 6 (Unique Decomposition) *If $\vdash e : \pi, \varphi$, then either*

1. *e is a value; or*
2. *there exists a unique $E_{\varphi'}$, ι, and π' such that $e = E_{\varphi'}[\iota]$ and $\vdash \iota : \pi', \varphi \cup \varphi'$.*

Proof. By induction on the structure of e. Suppose e is not a value. There are 9 cases to consider. We proceed by case analysis.

Case $e = d$ `at` ρ. It follows immediately that $E_{\varphi'} = [\cdot]$, $\iota = d$ `at` ρ, $\pi' = \pi$, and $\varphi' = \emptyset$.

Case $e =$ `letregion` ρ `in` e_1. A derivation $\vdash e : \pi, \varphi$ must end in a use of (13) followed by a number of uses of (5). It follows that there exists φ_1 and φ_2 such that $\varphi = \varphi_1 \setminus \{\rho\} \cup \varphi_2$ and $\rho \notin \text{frv}(\pi)$ and $\vdash e_1 : \pi, \varphi_1$. By renaming of bound variables, we can assume $\rho \notin \text{frv}(\varphi_2)$. By induction, either e_1 is a value or there exists a unique $E'_{\varphi''}$, ι_1, and π'_1 such that $e_1 = E'_{\varphi''}[\iota_1]$ and $\vdash \iota_1 : \pi'_1, \varphi_1 \cup \varphi''$. If e_1 is not a value then we take $E_{\varphi'} =$ `letregion` ρ `in` $E'_{\varphi''}$, $\varphi' = \varphi'' \cup \{\rho\}$, $\iota = \iota_1$, $\pi' = \pi'_1$, and from (5), we have $\vdash \iota_1 : \pi'_1, \varphi \cup \varphi'$, because $\varphi_1 \cup \varphi'' \subseteq \varphi \cup \varphi'$. Otherwise, $e_1 = v_1$ for some value v_1. Thus, $E_{\varphi'} = [\cdot]$, $\iota =$ `letregion` ρ `in` v_1, $\pi' = \pi$, and $\varphi' = \emptyset$.

Case $e = e_1\ e_2$. A derivation $\vdash e : \pi, \varphi$ must end in a use of (10), followed by a number of uses of (5). It follows that there exists μ, φ_1, φ_2, μ', ϵ, φ_0, and φ_3 such that $\varphi = \varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\} \cup \varphi_3$ and $\vdash e_1 : (\mu \xrightarrow{\epsilon.\varphi_0} \mu', \rho), \varphi_1$ and $\vdash e_2 : \mu, \varphi_2$ and $\pi = \mu'$. By induction, either e_1 is a value or else there exists $E'_{\varphi'_1}$, ι_1, and π'_1 such that $e_1 = E'_{\varphi'_1}[\iota_1]$ and $\vdash \iota_1 : \pi'_1, \varphi_1 \cup \varphi'_1$. If e_1 is not a value, then we take $E_{\varphi'} = E'_{\varphi'_1}\ e_2$, $\iota = \iota_1$, $\pi' = \pi'_1$, and because $\varphi' = \varphi'_1$ and $\varphi_1 \subseteq \varphi$, we can apply (5) to get $\vdash \iota_1 : \pi'_1, \varphi \cup \varphi'$. Otherwise, $e_1 = v_1$ for some value v_1. We can now apply the induction hypothesis to get that either e_2 is a value or else there exists $E'_{\varphi'_2}$, ι_2, and π'_2 such that $e_2 = E'_{\varphi'_2}[\iota_2]$ and $\vdash \iota_2 : \pi'_2, \varphi_2 \cup \varphi'_2$. If e_2 is not a value, then we take $E_{\varphi'} = v_1\ E'_{\varphi'_2}$, $\iota = \iota_2$, $\pi' = \pi'_2$, and because $\varphi' = \varphi'_2$ and $\varphi_2 \subseteq \varphi$, we can apply (5) to get $\vdash \iota_2 : \pi'_2, \varphi \cup \varphi'$. Otherwise $e_2 = v_2$ for some value v_2. Because $\vdash v_1 : (\mu \xrightarrow{\epsilon.\varphi_0} \mu', \rho), \varphi_1$, we can apply Proposition 5 to get $v_1 = \lambda x.e'$ `in` ρ. Thus, $E_{\varphi'} = [\cdot]$, $\varphi' = \emptyset$, $\iota = (\lambda x.e'$ `in` $\rho)\ v_2$, and $\pi' = \pi$.

The remaining cases follow similarly. □

Proposition 7 (Type Preservation) *If $\vdash e : \pi, \varphi$ and $e \overset{\varphi}{\longmapsto} e'$ then $\vdash e' : \pi, \varphi$.*

PROOF. By induction on the structure of e. We proceed by case analysis.

CASE $e = d$ `at` ρ. From assumptions and (6), we have $\pi = (\texttt{int}, \rho)$, and $\varphi = \{\rho\}$. From (16), we have $\rho \in \varphi$ and $e' = d$ `in` ρ. By use of (1) and (5), we have $\vdash e' : \pi, \varphi$, as required.

CASE $e =$ `letregion` ρ `in` v. From assumptions and from (13), there exists φ' such that $\varphi = \varphi' \setminus \{\rho\}$ and $\vdash v : \mu, \varphi'$. It follows from Proposition 2 that $\vdash v : \mu, \emptyset$, thus, from (20) and (5), we have $\vdash e' : \mu, \varphi$, as required.

CASE $e = (\lambda x.e_1$ `in` $\rho)\ v$. From assumptions, (10), and (3), there exists μ_1, ϵ, and φ_0 such that $\{x : \mu_1\} \vdash e_1 : \mu, \varphi_0$, $\vdash v : \mu_1, \varphi_1$, and $\varphi = \varphi_0 \cup \{\epsilon, \rho\}$. From Proposition 2, we have $\vdash v : \mu_1, \emptyset$. Thus, from Proposition 4, we have $\vdash e_1[v/x] : \mu, \varphi_0$. Now, because $\varphi \supseteq \varphi_0$, we can apply (5) to get $\vdash e' : \mu, \varphi$, as required.

CASE $e =$ (`fix` f `[`$\vec{\rho}$`]` x `=` e_1 `in` ρ) `[`$\vec{\rho}'$`]` `at` ρ'. From assumptions, (9), and (14), we have $\pi = (\tau, \rho')$, $\varphi = \{\rho, \rho'\}$, $v =$ `fix` f `[`$\vec{\rho}$`]` x `=` e_1 `in` ρ, $\sigma = \forall\vec{\rho}\vec{\alpha}\vec{\epsilon}.\mu_1 \xrightarrow{\epsilon.\varphi_0} \mu_2$, and

$$\vdash v : (\sigma, \rho), \emptyset \tag{67}$$

$$\sigma' = \forall\vec{\epsilon}\vec{\rho}.\mu_1 \xrightarrow{\epsilon.\varphi_0} \mu_2 \tag{68}$$

$$\sigma \geq \tau \text{ via } \vec{\rho}' \tag{69}$$

$$\{f : (\sigma', \rho)\}, x : \mu_1\} \vdash e_1 : \mu_2, \varphi_0 \tag{70}$$

From (67), (68), and (14), we have

$$\vdash v : (\sigma', \rho), \emptyset \tag{71}$$

From Proposition 4 and (71) and (70), we have

$$\{x : \mu_1\} \vdash e_1[v/f] : \mu_2, \varphi_0 \tag{72}$$

From the definition of generalization and from (69), there exists a substitution $S = ([\vec{\rho}'/\vec{\rho}], S^{\mathrm{t}}, S^{\mathrm{e}})$ such that

$$S(\mu_1 \xrightarrow{\epsilon.\varphi_0} \mu_2) = \tau \tag{73}$$

From (72) and (7), we have

$$\vdash \lambda x.e_1[v/f] \texttt{ at } \rho' : (\mu_1 \xrightarrow{\epsilon.\varphi_0} \mu_2, \rho'), \{\rho'\} \tag{74}$$

By renaming of bound names, we can assume $S(v) = v$ and $S(\rho') = \rho'$, thus, from (73), (74), and Proposition 1, we have $\lambda x.e_1[\vec{\rho}'/\vec{\rho}][v/f]$ `at` $\rho' : (\tau, \rho'), \{\rho'\}$. We can now apply (5) to get $\vdash e' : \pi, \varphi$, as required.

CASE $e =$ `#1` (v_1, v_2). From assumptions, (12), and (2), we have $\vdash v_1 : \mu, \emptyset$. We can now apply (5) to get $\vdash v_1 : \mu, \varphi$, as required.

CASE $e = E_{\varphi'}[e'']$. We have $e'' \overset{\varphi \cup \varphi'}{\longmapsto} e'''$ and $\varphi \cap \varphi' = \emptyset$ and $e' = E_{\varphi'}[e''']$. We now proceed by case analysis on the structure of $E_{\varphi'}$.

case $E_{\varphi'}[e''] =$ `(`e'', e_2`)` `at` ρ. We have $\varphi' = \emptyset$. From assumptions and (11) we have $\vdash e'' : \mu_1, \varphi_1$, $\vdash e_2 : \mu_2, \varphi_2$, $\mu = (\mu_1 \times \mu_2, \rho)$, and $\varphi = \varphi_1 \cup \varphi_2 \cup \{\rho\}$. By applying (5), we have $\vdash e'' : \mu_1, \varphi$. We can now apply the induction hypothesis to get $\vdash e''' : \mu_1, \varphi$. By applying (11), we have $\vdash E_{\varphi'}[e'''] : \mu, \varphi$, as required.

case $E_{\varphi'}[e''] =$ `(`v_1, e''`)` `at` ρ. We have $\varphi' = \emptyset$. From assumptions and (11) we have $\vdash v_1 : \mu_1, \varphi_1$, $\vdash e'' : \mu_2, \varphi_2$, $\mu = (\mu_1 \times \mu_2, \rho)$, and $\varphi = \varphi_1 \cup \varphi_2 \cup \{\rho\}$. By applying (5), we have $\vdash e'' : \mu_2, \varphi$. We can now apply the induction hypothesis to get $\vdash e''' : \mu_2, \varphi$. By applying (11), we have $\vdash E_{\varphi'}[e'''] : \mu, \varphi$, as required.

case $E_{\varphi'}[e''] =$ `#`$i\ e''$, $i \in \{1, 2\}$. We have $\varphi' = \emptyset$. From assumptions and (12), we have $\vdash e'' : (\mu_1 \times \mu_2, \rho), \varphi'$, $\mu = \mu_i$ and $\varphi = \varphi' \cup \{\rho\}$. By applying (5), we have $\vdash e'' : (\mu_1 \times \mu_2, \rho), \varphi$, thus, we can apply the induction hypothesis to get $\vdash e''' : (\mu_1 \times \mu_2, \rho), \varphi$. We can now apply (12) to get $\vdash E_{\varphi'}[e'''] : \mu, \varphi$, as required.

case $E_{\varphi'}[e''] =$ `let` x `=` e'' `in` e_2. We have $\varphi' = \emptyset$. From assumptions and (15), there exists π such that $\vdash e'' : \pi, \varphi_1$, $\{x : \pi\} \vdash e_2 : \mu, \varphi_2$, and $\varphi = \varphi_1 \cup \varphi_2$. Applying (5), we have $\vdash e'' : \pi, \varphi$. By induction, we have $\vdash e''' : \pi, \varphi$. We can now apply (15) to get $\vdash E_{\varphi'}[e'''] : \mu, \varphi$, as required.

case $E_{\varphi'}[e''] = e''\ e_2$. From assumptions and (10), there exists $\epsilon, \varphi_0, \varphi_1, \varphi_2$, and ρ such that $\vdash e'' : (\mu_2 \xrightarrow{\epsilon.\varphi_0} \mu, \rho), \varphi_1$, $\vdash e_2 : \mu_2, \varphi_2$, and $\varphi = \varphi_0 \cup \varphi_1 \cup \varphi_2 \cup \{\epsilon, \rho\}$. From (5), we have $\vdash e'' : (\mu_2 \xrightarrow{\epsilon.\varphi_0} \mu, \rho), \varphi$, thus, by induction, we have $\vdash e''' : (\mu_2 \xrightarrow{\epsilon.\varphi_0} \mu, \rho), \varphi$. We can now apply (10) to get $\vdash E_{\varphi'}[e'''] : \mu, \varphi$, as required.

case $E_{\varphi'}[e''] = v\ e''$. As above.

case $E_{\varphi'}[e''] = e''$ `[`$\vec{\rho}$`]` `at` ρ. As above.

case $E_{\varphi'}[e''] =$ `letregion` ρ `in` e''. We have $\varphi' = \{\rho\}$. From assumptions and from (13), there exists φ'' such that $\varphi = \varphi'' \setminus \{\rho\}$, and $\vdash e'' : \mu, \varphi''$. From (5), we have $\vdash e'' : \mu, \varphi \cup \varphi'$. We can now apply the induction hypothesis to get $\vdash e''' : \mu, \varphi \cup \varphi'$. Now, because $\varphi = (\varphi \cup \varphi') \setminus \{\rho\}$, we can apply (13) to get $\vdash E_{\varphi'}[e'''] : \mu, \varphi$, as required.

The remaining cases follow similarly. □

PROPOSITION 8 (PROGRESS) *If* $\vdash e : \pi, \varphi$ *then either* e *is a value or else there exists some* e' *such that* $e \overset{\varphi}{\longmapsto} e'$.

PROOF. If e is not a value, then by Proposition 6 there exists a unique $E_{\varphi'}$, ι, and π' such that $e = E_{\varphi'}[\iota]$ and $\vdash \iota : \pi', \varphi \cup \varphi'$. We argue that $\iota \overset{\varphi \cup \varphi'}{\longmapsto} e_2$, for some e_2, so that $E_{\varphi'}[\iota] \overset{\varphi}{\longmapsto} E_{\varphi'}[e_2]$ follows from (26). We now consider all cases where ι could possibly be stuck.

CASE $\iota = d$ `at` ρ. From (6) and (5), we have $\rho \in \varphi \cup \varphi'$. We can now apply (16) to get $e_2 = d$ `in` ρ.

CASE $\iota = (\lambda x.e_{\mathrm{x}}$ `in` $\rho)\ v$. We have $\vdash (\lambda x.e_{\mathrm{x}}$ `in` $\rho)\ v : \pi', \varphi \cup \varphi'$. This derivation must end in an application of (10) followed by a number of applications of (5). Thus, by applying Proposition 2, there exists μ, μ', ϵ, and φ_0 such that $\vdash \lambda x.e_{\mathrm{x}}$ `in` $\rho : (\mu \xrightarrow{\epsilon.\varphi_0} \mu', \rho), \emptyset$ and $\vdash v : \mu, \emptyset$ and $\pi' = \mu'$ and $\varphi_0 \cup \{\epsilon, \rho\} \subseteq \varphi \cup \varphi'$. Now, because $\rho \in \varphi \cup \varphi'$, we can apply (21) to get $e_2 = e_{\mathrm{x}}[v/x]$.

CASE $\iota =$ (`fix` f `[`$\vec{\rho}$`]` x `=` e_0 `in` ρ') `[`$\vec{\rho}'$`]` `at` ρ. The derivation $\vdash \iota : \pi', \varphi \cup \varphi'$ must end in an application of (9) followed by a number of applications of (5), thus, using Proposition 2, there exist σ and τ' such that $\pi' = (\tau', \rho)$ and

$$\vdash \texttt{fix } f \texttt{ [}\vec{\rho}\texttt{] } x \texttt{ = } e_0 \texttt{ in } \rho' : (\sigma, \rho'), \emptyset \tag{75}$$

$$\sigma \geq \tau' \text{ via } \vec{\rho}' \tag{76}$$

$$\{\rho, \rho'\} \subseteq \varphi \cup \varphi' \tag{77}$$

By the definition of generalization and from (76), we have $\sigma = \forall\vec{\rho}\vec{\alpha}\vec{\epsilon}.\tau$, for some $\vec{\alpha}$, $\vec{\epsilon}$, and τ. Now, because $\rho' \in \varphi \cup \varphi'$ follows from (77), we can apply (23) to get $e_2 = \lambda x.e_0[\vec{\rho}'/\vec{\rho}][v/f]$ `at` ρ, where $v =$ `fix` f `[`$\vec{\rho}$`]` x `=` e_0 `in` ρ'.

CASE $\iota =$ `letregion` ρ `in` v. It follows immediately from (20) that $e_2 = v$.

The remaining cases follow similarly. □