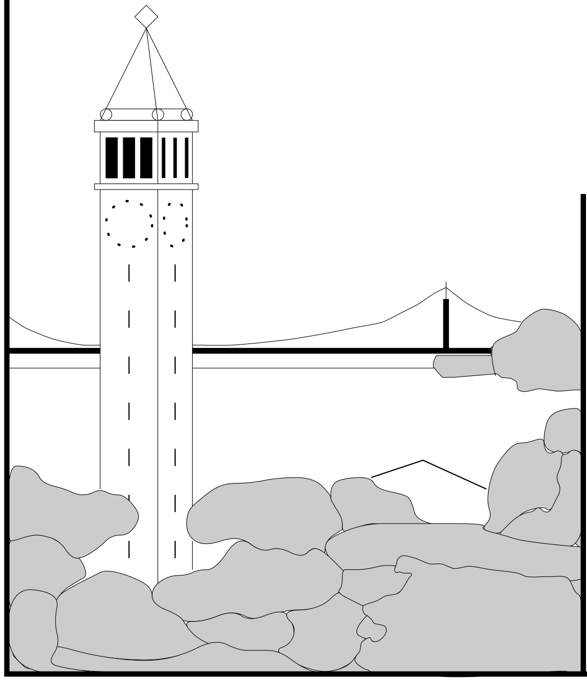
Memory Management with Use-Counted Regions

Tachio Terauchi *Alex Aiken*



Report No. UCB/CSD-4-1314

March 2004

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Memory Management with Use-Counted Regions [*]

Tachio Terauchi   Alex Aiken
EECS Department
University of California, Berkeley
Berkeley, CA 94720-1776
tachio@cs.berkeley.edu

March 29, 2004

Abstract

We introduce a new region-based memory management technique that allows flexible memory usage patterns and is provably safe. Our technique is explicit and manual like C's `malloc` and `free`, and it allows programmers to exert a similar degree of control over the program behavior. Our method is quite simple in spite of this expressiveness and safety.

We apply the technique to a small functional language to formally describe the core concepts, prove its safety, and argue its usability and efficiency analytically. In particular, we show that the system can efficiently encode more rigid, traditional regions whose lifetime is bounded by that of a stack frame. We also show that the technique works nicely with multi-threaded and imperative programs.

1 Introduction

In software practice, there are two widely-used heap memory management approaches: explicit management via `malloc` and `free` or implicit management via garbage collection. Both approaches have well-known disadvantages. The problem with `malloc` and `free` is that verifying memory safety is difficult. Garbage collection overcomes this problem at the cost of making it difficult to control when objects are deallocated, which makes garbage collection unattractive in situations that require tight space or timing guarantee. In addition, experienced programmers can often produce more efficient code given the ability to do manual memory management. Thus a memory management system that allows programmers to exert manual control and yet be provably safe is of practical interest.

Recently, there have been a number of proposals [4, 18, 9, 2, 11] based on the concept of regions [14, 15], that allow safe, explicitly controlled memory management. All these systems (with the exception of portions of [11] which was developed independently from our work around the same time) guarantee safety at compile time.

This paper presents a new *mostly-static*, memory-safe region system. It is not *totally static* in the same sense as the systems listed above because the deletion of a region could fail and result in a run-time error, and the programmer is required to add run-time checks to see if regions are alive at strategic points in the program. The system is *mostly static* because it guarantees at compile time that no other run-time checks are necessary for the program to be memory safety. Also, unlike dangling pointer dereferences, our run-time errors are well-behaved and the program can recover by, for example, catching the thrown exception ala ML and Java.

[*]This research was supported in part by Subcontract no. PY-1099 to Stanford, Dept. of the Air Force prime contract no. F33615-00-C-1693.

$\mathtt{U}, \mathtt{V}, \mathtt{W}, \mathtt{X}, \mathtt{Y}, \mathtt{Z} \in \text{RegVars} \cup \{\mathtt{X}_{\mathtt{sr}}\}$ $\mathtt{u}, \mathtt{v}, \mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z} \in \text{Vars} \cup \{\mathtt{sr}\}$ $L \subseteq_{\text{finite}} \text{RegVars} \cup \{\mathtt{X}_{\mathtt{sr}}\}$ $\mathtt{r}, \mathtt{s}, \mathtt{t} \in \text{Regions}$

$$\begin{array}{lll}
\text{types} & \tau ::= & (\forall \vec{\mathtt{X}}.\tau_1 \xrightarrow{L} \tau_2)@\mathtt{Y} \mid \vec{\tau}@\mathtt{X} \mid \mathit{reg}(\mathtt{X})@\mathtt{Y} \mid \exists \mathtt{X}.\tau \\
\text{expressions} & e ::= & \mathtt{x} \mid \mathtt{let}\ \mathtt{x} = e_1\ \mathtt{in}\ e_2 \mid \Lambda\vec{\mathtt{X}}\lambda\mathtt{x}{:}\tau.e_1\ \mathtt{at}\ e_2 \mid e_1[\vec{\mathtt{X}}]\ e_2 \mid \vec{e}\ \mathtt{at}\ e \mid e.i \mid \mathtt{pack}\ e\ \mathtt{as}\ \exists\mathtt{X}.\tau \mid \mathtt{open}\ \mathtt{x} = e_1\ \mathtt{as}\ \tau\ \mathtt{in}\ e_2 \mid \\
 & & \mathtt{newregion}\ \mathtt{at}\ e \mid \mathtt{freeregion}\ e \mid \mathtt{useregion}\ e_1\ \mathtt{in}\ e_2
\end{array}$$

Figure 1: Small region language: source syntax

Compared to a totally-static approach, our system has a slightly weaker memory-safety guarantee.[1] But we believe that there are enough benefits with our mostly-static approach to justify the trade-off. One drawback of a totally static approach is that it often trades simplicity and ease-of-use for expressiveness because a simple static system often limits the kind of programs that can be proven safe. In contrast, our system allows a high degree of expressiveness without complicating the static system. In particular, a region's lifetime is not tied to a stack frame, and instead regions may be created and deleted at any program point. We also argue in Section 4.2 that a run-time error is likely to indicate a bug in the programmer's thinking rather than a limitation of the system.

Our technique is best understood by conceptually dividing it into a static part and a dynamic part. The static part is simple, based on a type system resembling [15]. The dynamic part is simple for the most basic implementation, which we shall stick with until Section 3. The run-time behavior is transparent to the programmer. That is, the programmer can easily tell from the source code when each dynamic check occurs, which is important for control.

We now describe the core concepts of our technique, which we call *use counting*. (We shall slightly modify these concepts in Section 3 to enable further flexibility). The run-time system maintains two data values per region:

- *Validity bit*: indicating whether region has been deleted (= 0) or not (= 1).
- *Use counter*: a non-negative integer which starts from 0 and is incremented and decremented explicitly by the program.

Via a combination of static and dynamic checks, we enforce the following:

(1) When accessing a memory location, the use counter of the region containing the memory location is greater than 0.

(2) When deleting a region, the region's use counter is 0.

(3) When incrementing the use counter of a region, the region's validity bit is 1.

It is not hard to see that these conditions are sufficient for *memory safety*: dangling references are never accessed. We check the condition (1) statically and check (2) and (3) dynamically. Since deletions are explicit, dynamic checks for (2) are transparent. Incrementing a use counter is also explicit, so dynamic checks for (3) are also transparent.

The trick, then, is to make the static check for (1) simple. To this end, we design the language so that programs adhere to a certain syntactic pattern when incrementing and decrementing use counters. Consider the following pseudo-code. Suppose `r` denotes some region.

```
increment_uc(r)
... use r ...
decrement_uc(r)
```

[1]Though in principle, any well-behaved run-time error is a part of the program semantics, and therefore we could argue that our system is just as static.

values	v	::=	`pack` v `as` • | Λ•λ`x`:•.e `at r` | $\vec{v}$ `at r` | `reg r at s` | `abort`
expressions	e	::=	`x` | `let x =` e_1 `in` e_2 | Λ•λ`x`:•.e_1 `at` e_2 | e_1[•] e_2 | $\vec{e}$ `at` e | $e.i$ | `pack` e `as` • | `open x =` e_1 `as` • `in` e_2 | `newregion at` e | `freeregion` e | `useregion` e_1 `in` e_2 | v | `inuse (reg r at s) in` e
contexts	E	::=	[ ] | `let x =` E `in` e | Λ•λ`x`:•.e `at` E | E[•] e | v[•] E | ($\vec{v}, E, \vec{e}$) `at` e | $\vec{v}$ `at` E | `pack` E `as` • | `open x =` E `as` • `in` e | `newregion at` E | `freeregion` E | `useregion` E `in` e | `inuse (reg r at s) in` E

Figure 2: Small region language: type-erased intermediate expressions and evaluation contexts

The operations `increment_uc(r)` and `decrement_uc(r)` respectively increment and decrement the use counter of `r`. Our proposal is to force programs to adhere to this block pattern when manipulating a use counter. To this end, instead of `increment_uc(r)` and `decrement_uc(r)`, we introduce a single language construct for increment and decrement:

`useregion` e_1 `in` e_2

which evaluates e_1 to a use counter, increments it, executes e_2, then decrements the use counter after e_2 finishes. From a programmer's point of view, `useregion` enables access to the region within the lexical scope. The resulting system is simple and easy to use, like stack-of-regions systems [9, 2], but enjoys expressiveness comparable to more complex approaches.

1.1 Contributions and Overview

The main technical contributions of the paper are:

- a formal presentation of the basic use-counting system and the proof of safety (Section 2).
- an improvement to the base system: "use counters for free" (Section 3). This feature is not introduced immediately in the paper since it is more natural to understand it as a modification to the base system.
- an analytical evaluation of the system, including efficient encoding of stack-of-regions style memory management and multi-threaded programming (Sections 4 and 5).

In addition to the above, we provide small but illustrative examples throughout the paper. Section 6 discusses our own experience with a toy implementation. Section 7 discusses related work, and Section 8 concludes.

2 Regions for a Small Language

We first apply use counting to a call-by-value monomorphic lambda calculus with tuples shown in Figure 1. This small region language is designed to illustrate the new technique in a self-contained manner and therefore omits features such as parametric polymorphism over types and effect sets, recursive types, non-heap allocated tuples, and mutable values. But these features can be incorporated with little additional effort by leveraging off previous research on region-based memory management (see, e.g., [8] for an extensive study incorporating these features and more in a stack-of-regions style framework). Appendix A shows the system extended with some of these features.

RegVars is an infinite set of region variables. Each region variable refers to some region. For example, a function closure allocated in the region referred to by `Y` has a type of the form $(\forall\vec{\mathtt{X}}.\tau_1 \xrightarrow{L} \tau_2)$@`Y` with τ_1 as the argument type and τ_2 as the return type. Note that the function may be polymorphic in the region variables $\vec{\mathtt{X}}$, which are assumed to be distinct. (We use the notation $\vec{a}$ to mean some tuple $(a_1, a_2, ..., a_n)$.) In addition, each function type carries a set of region variables, an *effect set* L, referring to a superset of the regions the function accesses. Similarly, a tuple of values (of the types $\vec{\tau}$) allocated in `X` has the type $\vec{\tau}$@`X`. The type *reg*(`X`)@`Y` is the type of a *region handle* of the region referred to by `X`. A region handle is a program

$$\frac{R(\mathtt{r}) = (1,-) \qquad R(\mathtt{s}) = (1,-)}{R, E[\Lambda\bullet\lambda\mathtt{x}{:}\bullet.e\ \mathtt{at}\ (\mathtt{reg\ r\ at\ s})] \to R, E[\Lambda\bullet\lambda\mathtt{x}{:}\bullet.e\ \mathtt{at\ r}]}\ [\text{F1}] \qquad \frac{R(\mathtt{r}) = (1,-)}{R, E[(\Lambda\bullet\lambda\mathtt{x}{:}\bullet.e\ \mathtt{at\ r})[\bullet]\ v] \to R, E[e[v/\mathtt{x}]]}\ [\text{F2}]$$

$$\frac{}{R, E[\mathtt{open\ x\ =\ (pack}\ v\ \mathtt{as}\ \bullet)\ \mathtt{as}\ \bullet\ \mathtt{in}\ e] \to R, E[e[v/\mathtt{x}]]}\ [\text{E}] \qquad \frac{R(\mathtt{r}) = (1,-) \qquad R(\mathtt{s}) = (1,-)}{\begin{array}{l} R, E[\mathtt{newregion\ at\ (reg\ r\ at\ s)}] \\ \quad \to R \uplus \{\mathtt{t} \mapsto (1,0)\}, E[\mathtt{pack\ (reg\ t\ at\ r)\ as}\ \bullet] \end{array}}\ [\text{R1}]$$

$$\frac{R(\mathtt{s}) = (1,-)}{\begin{array}{l} R \uplus \{\mathtt{r} \mapsto (-,0)\}, E[\mathtt{freeregion\ (reg\ r\ at\ s)}] \\ \qquad\qquad \to R \uplus \{\mathtt{r} \mapsto (0,0)\}, E[\mathtt{reg\ r\ at\ s}] \end{array}}\ [\text{R2}] \qquad \frac{R(\mathtt{s}) = (1,-) \qquad i > 0}{\begin{array}{l} R \uplus \{\mathtt{r} \mapsto (-,i)\}, E[\mathtt{freeregion\ (reg\ r\ at\ s)}] \\ \qquad\qquad \to R \uplus \{\mathtt{r} \mapsto (-,i)\}, \mathtt{abort} \end{array}}\ [\text{R3}]$$

$$\frac{R(\mathtt{s}) = (1,-)}{R \uplus \{\mathtt{r} \mapsto (1,i)\}, E[\mathtt{useregion\ (reg\ r\ at\ s)\ in}\ e] \to R \uplus \{\mathtt{r} \mapsto (1,i+1)\}, E[\mathtt{inuse\ (reg\ r\ at\ s)\ in}\ e]}\ [\text{R4}]$$

$$\frac{R(\mathtt{s}) = (1,-)}{\begin{array}{l} R \uplus \{\mathtt{r} \mapsto (0,i)\}, E[\mathtt{useregion\ (reg\ r\ at\ s)\ in}\ e_2] \\ \qquad\qquad \to R \uplus \{\mathtt{r} \mapsto (0,i)\}, \mathtt{abort} \end{array}}\ [\text{R5}] \qquad \frac{R(\mathtt{s}) = (1,-)}{\begin{array}{l} R \uplus \{\mathtt{r} \mapsto (b,i)\}, E[\mathtt{inuse\ (reg\ r\ at\ s)\ in}\ v] \\ \qquad\qquad \to R \uplus \{\mathtt{r} \mapsto (b,i-1)\}, E[v] \end{array}}\ [\text{R6}]$$

Figure 3: Small region language: selected reduction rules

value, so it is allocated in a region just like other program values. Region handles are necessary because the validity bit and use counter for each region must be stored somewhere; that is, each region handle is a pointer to a triple containing a pointer to the head of the region, a validity bit, and a use counter.[2] The type $\exists\mathtt{X}.\tau$ is an existential type, which is handy when typing data structures containing region handles. (In theory, we can always encode them as higher-order polymorphic functions, but existential packages are more natural.) We assume that expressions and types are equivalent up to consistent renaming of bound variables and bound region variables.

A *program* is an expression whose only free variable is `sr`. The set of free variables for an expression is defined in the usual way. The variable `sr` denotes a region handle of a special region, $\mathtt{r_{sr}}$. (It doesn't matter which region we pick to be special; we just need to pick one from Regions, an infinite set of regions, before the program starts.) The region $\mathtt{r_{sr}}$ is special because it is the only region that exists prior to the execution of the program; all other regions are created by the program. To see why we need this starting region, recall that each region handle must itself be allocated somewhere. The core concepts described in Section 1 implies that in order for a region to be deleted, its region handle cannot be allocated within the region itself. By chain of reasoning, there must be at least one pre-existing region in which the first program-created region handle is allocated. We shall remove this constraint in Section 3.

2.1 Dynamic Semantics

The dynamic semantics is a series of small-step reductions from states to states. A *state* is a pair (R, e) where e is a program and R is a *region map*, which maps each region $\mathtt{r}$ in its domain to a pair (b, i) with $\mathtt{r}$'s validity bit b and $\mathtt{r}$'s use counter i. The starting state for the program e is $(\{\mathtt{r_{sr}} \mapsto (\mathtt{1},\mathtt{1})\}, \mathtt{e})$. Note that the starting region is accessible from the start. In fact, the starting region cannot be deleted and will remain always accessible, so the program never needs to manipulate its use counter. Single step reductions are written

$$R_1, e_1 \to R_2, e_2$$

(We usually omit the parentheses.) Type and region variable information is irrelevant in reductions, so we erase it from expressions by replacing each occurrence of a type or a region variable by $\bullet$. The result of this erasure is an expression in the language of Figure 2.

[2]Alternatively, we can implement a region handle as a double-word value consisting of a pointer to the head of the region and a pointer to the pair containing the validity bit and the use counter to save dynamic checks at allocation sites.

Figure 3 shows selected reduction rules. Reductions may give rise to expressions in the intermediate language and use evaluation contexts defined in Figure 2. We now discuss the reduction rules. To reduce a function allocation $\Lambda\bullet\lambda\texttt{x}{:}\bullet.e_1$ `at` e_2, first e_2 is reduced to a region handle `reg r at s`. Then [F1] reduces the whole expression to a function closure $\Lambda\bullet\lambda\texttt{x}{:}\bullet.e_1$ `at r`. The expression `reg r at s` is a region handle for the region `r` allocated in the region `s`. Every region handle is represented in this form. (So before the program starts reducing, we replace every occurrence of `sr` by `reg` $\texttt{r}_{\texttt{sr}}$ `at` $\texttt{r}_{\texttt{sr}}$.) In fact, every allocated program value has the form

$$a \; \texttt{at r}$$

meaning the value denoted by a is allocated in the region `r`. So, $\Lambda\bullet\lambda\texttt{x}{:}\bullet.e_1$ `at r` is a function closure value allocated in `r`.[3] The condition $R(\texttt{r}) = (1, -)$ of [F1] says that the region where the function is being allocated must be alive. (The symbol $-$ is a wild card.) In addition, the condition $R(\texttt{s}) = (1, -)$ is required because a region handle also stores a pointer to the head of the region and this needs to be looked up.

The rule [F2] is for function application. It shows function application as the usual capture-avoiding substitution assuming `r`, the region where the function is allocated, is alive.

The rule [E] is for opening existential packages and is straightforward.

Once e in `newregion at` e is reduced to a region handle `reg r at s`, [R1] reduces `newregion at (reg r at s)` by creating a new region `t` and reducing to a region handle of `t` allocated at `r`. The binary operation $R_1 \uplus R_2$ is the union $R_1 \cup R_2$ if $dom(R_1) \cap dom(R_2) = \emptyset$ and undefined otherwise. For reasons explained later, the new region handle is existentially packed. The conditions $R(\texttt{r}) = (1, -)$ and $R(\texttt{s}) = (1, -)$ are required because allocating a region handle is like allocating any other program value, that is, the region where the region handle is allocated must be alive as well as the region where the region handle itself resides.

The rules [R2] and [R3] handle `freeregion` e. We first reduce e to a region handle. Assuming that the use counter of the region being deleted is 0, [R2] sets the validity bit of the region to 0, indicating that the region is deleted. Otherwise the use counter of the region is greater than 0, and [R3] reduces the whole program to the special value `abort` which signals a failure of the dynamic check. Alternatively, the language could have been designed so that a dynamic check failure reduces to a special error value or raises a run-time exception. Exactly what to do when a dynamic check fails is up to the language designer. Note that both [R2] and [R3] require the assumption $R(\texttt{s}) = (1, -)$ because the region handle for `r` allocated at `s` must be looked up to see which of [R2] or [R3] applies.

The form `useregion` e_1 `in` e_2 increments and decrements use counters. First, e_1 is reduced to a region handle of some region `r`, then if `r`'s validity bit is 1, the use counter is incremented and the whole expression is reduced to `inuse (reg r at s) in` e_2 as shown in [R4]. Otherwise the region's validity bit is 0, and [R5] `abort`s the program. Both [R4] and [R5] require the assumption $R(\texttt{s}) = (1, -)$ because the use counter and the validity bit of `r` are accessed.

The rule [R6] reduces `inuse (reg r at s) in` e. First, e is reduced to the value v, then the use counter of `r` is decremented and v is returned, assuming that the region where the use counter is allocated is alive.

Note that the run-time system does not physically check the conditions of the form $R(\texttt{r}) = (1, -)$ appearing in any of the hypotheses in Figure 3. Violation of these conditions correspond to dangling pointer dereferences, and we will argue in Section 2.4 that well-typed programs never violate them.

However, a cautious reader may wonder how we can make safety claims by assuming that the validity bits even exist in these hypothesis. What if the validity bit is deallocated prior to the reduction? To see that such situation never occurs, observe that the dynamic semantics never deallocates any value; it just sets some validity bits to 0. In order to actually deallocate, we first argue as in the previous paragraph that the conditions of the form $R(\texttt{r}) = (\texttt{1}, -)$ are not required if the program is known not to get stuck, which in turn lets us physically delete regions at [R2] as argued in Section 2.4.

[3]This a; `at r` notation may look strange to some readers. More commonly, when an expression allocates a program value, the expression reduces to a pointer pointing to the memory location where the program value is stored. But because every program value in the language except region handles is immutable, we save a few characters by not showing the extra level of indirection. If some readers are uncomfortable with our notation, they may, for each state, imagine a store partitioned into regions and replace each a; `at r` with a pointer to the memory location containing a in the `r` partition of the store. Note that region handles are mutable because the validity bit and the use counter may change. For this reason, we have region maps to handle the extra level of indirection.

$$\frac{\Gamma(\mathtt{x}) = \tau}{\Delta;\Gamma \vdash \mathtt{x} : \tau; L}\ \text{(V)} \qquad \frac{\Delta;\Gamma \vdash e_1 : \tau_1; L \quad \Delta;\Gamma \uplus \{\mathtt{x} \mapsto \tau_1\} \vdash e_2 : \tau_2; L}{\Delta;\Gamma \vdash \mathtt{let\ x\ =}\ e_1\ \mathtt{in}\ e_2 : \tau_2; L}\ \text{(L)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_2 : \mathit{reg}(\mathtt{X})@\mathtt{Z}; L_2 \quad \mathit{fregvars}(\tau_1) \cup L_1 \subseteq \Delta \uplus \{\vec{\mathtt{Y}}\} \\ \Delta \uplus \{\vec{\mathtt{Y}}\}; \Gamma \uplus \{x \mapsto \tau_1\} \vdash e_1 : \tau_2; L_1 \quad \mathtt{X}, \mathtt{Z} \in L_2\end{array}}{\Delta;\Gamma \vdash \Lambda\vec{\mathtt{Y}}\lambda\mathtt{x}{:}\tau_1.e_1\ \mathtt{at}\ e_2 : (\forall\vec{\mathtt{Y}}.\tau_1 \xrightarrow{L_1} \tau_2)@\mathtt{X}; L_2}\ \text{(F1)} \qquad \frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : (\forall\vec{\mathtt{Y}}.\tau_1 \xrightarrow{L_1} \tau_2)@\mathtt{Z}; L_2 \quad \{\vec{\mathtt{X}}\} \subseteq \Delta \\ \Delta;\Gamma \vdash e_2 : \tau_1[\vec{\mathtt{X}}/\vec{\mathtt{Y}}]; L_2 \quad L_1[\vec{\mathtt{X}}/\vec{\mathtt{Y}}] \cup \{\mathtt{Z}\} \subseteq \mathtt{L}_2\end{array}}{\Delta;\Gamma \vdash e_1[\vec{\mathtt{X}}]\ e_2 : \tau_2[\vec{\mathtt{X}}/\vec{\mathtt{Y}}]; L_2}\ \text{(F2)}$$

$$\frac{\Delta;\Gamma \vdash \vec{e} : \vec{\tau}; L \quad \Delta;\Gamma \vdash e : \mathit{reg}(\mathtt{X})@\mathtt{Y}; L \quad \mathtt{X}, \mathtt{Y} \in L}{\Delta;\Gamma \vdash \vec{e}\ \mathtt{at}\ e : \vec{\tau}@\mathtt{X}; L}\ \text{(T1)} \qquad \frac{\Delta;\Gamma \vdash e : (\tau_1, \tau_2, ..., \tau_i, ...)@\mathtt{X}; L \quad \mathtt{X} \in L}{\Delta;\Gamma \vdash e.i : \tau_i; L}\ \text{(T2)}$$

$$\frac{\Delta;\Gamma \vdash e : \tau; L}{\Delta;\Gamma \vdash \mathtt{pack}\ e\ \mathtt{as}\ \exists\mathtt{X}.\tau : \exists\mathtt{X}.\tau; L}\ \text{(E1)} \qquad \frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \exists\mathtt{X}.\tau_1; L \quad \mathit{fregvars}(\tau_1) \subseteq \Delta \uplus \{\mathtt{X}\} \\ \Delta \uplus \{\mathtt{X}\}; \Gamma \uplus \{\mathtt{x} \mapsto \tau_1\} \vdash \mathtt{e}_2 : \tau_2; \mathtt{L} \quad \mathit{fregvars}(\tau_2) \subseteq \Delta\end{array}}{\Delta;\Gamma \vdash \mathtt{open\ x\ =}\ e_1\ \mathtt{as}\ \tau_1\ \mathtt{in}\ e_2 : \tau_2; L}\ \text{(E2)}$$

$$\frac{\Delta;\Gamma \vdash e : \mathit{reg}(\mathtt{Y})@\mathtt{Z}; L \quad \mathtt{Y}, \mathtt{Z} \in L \quad \mathtt{X} \neq \mathtt{Y}}{\Delta;\Gamma \vdash \mathtt{newregion\ at}\ e : \exists\mathtt{X}.\mathit{reg}(\mathtt{X})@\mathtt{Y}; L}\ \text{(R1)} \qquad \frac{\Delta;\Gamma \vdash e : \mathit{reg}(\mathtt{X})@\mathtt{Y}; L \quad \mathtt{Y} \in L}{\Delta;\Gamma \vdash \mathtt{freeregion}\ e : \mathit{reg}(\mathtt{X})@\mathtt{Y}; L}\ \text{(R2)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \mathit{reg}(\mathtt{X})@\mathtt{Y}; L \quad \Delta;\Gamma \vdash e_2 : \tau; L \cup \{\mathtt{X}\} \quad \mathtt{Y} \in L}{\Delta;\Gamma \vdash \mathtt{useregion}\ e_1\ \mathtt{in}\ e_2 : \tau; L}\ \text{(R3)}$$

Figure 4: Small region language: type checking rules

2.2 Static Checking

As stated in Section 1, the role of the static checking is to check that when accessing a memory location, the use counter of the region in which the memory location is allocated is greater than 0. The use counter is incremented before reducing e_2 of `useregion` e_1 `in` e_2 and decremented after e_2 reduces to a value. Therefore the plan is to disallow accesses to a region `r` except in the context of some `useregion` e_1 `in` e_2 such that e_1 is a region handle of `r`.

We use a type and effect system [7, 12] where the type judgement $\Delta;\Gamma \vdash e : \tau; L$ reads e has the type τ in the environment $\Delta;\Gamma$ and the effect set L. In the above judgement, Δ, a *region variable environment*, is a set of region variables and Γ, a *type environment*, is a mapping from variables to types. Figure 4 shows the type checking rules.

We briefly discuss the type checking rules. In (V), note that any L including $\emptyset$ can appear in the conclusion because any expression substituted for `x` must be a value and therefore already reduced (because of call-by-value semantics).

The rule (L) for local variable definition is self-explanatory.

The rule (F1) types function allocation. The notation *fregvars*(τ) denotes the set of free region variables of τ. The function will be allocated in the region specified by the region handle e_2, referred to by the region variable `X`. The pointer to the region is stored in the region referred to by `Z`. Hence $\mathtt{X}, \mathtt{Z} \in L_2$ is required. The assumption $\Delta \uplus \{\vec{\mathtt{Y}}\}; \Gamma \uplus \{x \mapsto \tau_1\} \vdash e_1 : \tau_2; L_1$ types the body of the function. Note that L_1 also appears in the function type; this set is the *latent effect* of the function that takes place when the function is applied. Here, we have overloaded $\uplus$ such that $\Delta_1 \uplus \Delta_2$ is the standard disjoint union.

The rule (F2) types function application. The latent effect L_1 is carried in the type of the function, therefore $L_1[\vec{\mathtt{X}}/\vec{\mathtt{Y}}] \subseteq L_2$ is required because reducing the function application leads to reducing the body of the function. Furthermore, $\mathtt{Z} \in L_2$ is required because the function closure value is accessed.

The rule (T1) types tuple allocation. The judgement $\Delta;\Gamma \vdash \vec{e} : \vec{\tau}; L$ is a short-hand for $\Delta;\Gamma \vdash e_1 : \tau_1; L$, $\ldots$, $\Delta;\Gamma \vdash e_n : \tau_n; L$. Like in (F1), $\mathtt{X}, \mathtt{Y} \in L$ is required. The rule (T2) types tuple projection. The condition $\mathtt{X} \in L$ is required because the tuple is accessed.

The rules (E1) and (E2) type existential packaging and opening. Instead of explicit substitutions, we let region variables match via implicit renaming of bound region variables.[4]

The rule (R1) types region creation. The type of the expression is an existentially quantified region handle type, matching the reduction rule [R1]. Existential types statically distinguish different regions, that is, different regions are assigned different region variables if they appear free in a piece of code. For a reason similar to that in (F1) and (T1), the condition $\mathtt{Y}, \mathtt{Z} \in L$ is required.

The rule (R2) types region deletion. The condition $\mathtt{Y} \in L$ is required because the validity bit stored in the region referred to by Y must be set to 0 and the use counter needs to be checked.

The rule (R3) types manipulation of use counters. Since the use counter for the region referred to by X is incremented on entry into e_2, we allow e_2 to access X; hence e_2 is type-checked with the effect $L \cup \{\mathtt{X}\}$. The use counter may be decremented within e_2, but this does not pose a problem because it must be the case that decrementing happens at the end of reducing some $\mathtt{useregion} - \mathtt{in}\ e_3$ reachable by reducing e_2. Therefore the use counter must have been incremented on entry to e_3. Therefore it follows that the use counter at the end of reducing e_2 is at least as large as it was immediately before reducing e_2 (in this case, they are actually equal). This intuition works for the single-thread case. We shall formalize the argument later so that it works even for multi-thread programs.

Given a program e, we say that e is *well-typed* iff $\{\mathtt{X_{sr}}\}; \{\mathtt{sr} \mapsto \mathit{reg}(\mathtt{X_{sr}})@\mathtt{X_{sr}}\} \vdash e : \tau; \{\mathtt{X_{sr}}\}$ for some τ.

All the rules are syntax directed, and type checking is clearly decidable. Readers familiar with stack-of-regions style systems should be able to easily recognize all rules except for (R1), (R2), and (R3).

2.3 Example 1

The expression below shows a function to be allocated at the starting region. The function takes a region handle of any region r (referred to by X in the function body) as the argument, creates a new region, say s, whose region handle gets allocated in r, creates another new region, say t, whose region handle gets allocated in s, and returns both of the new region handles in a tuple allocated in r.

```
ΛX, Yλx: reg(X)@Y.
      open z = (newregion at x) as reg(Z)@X
      in useregion z
         in pack (z, newregion at z) at x
            as ∃Z.(reg(Z)@X, ∃W.reg(W)@Z)@X
at sr
```

The expression is well-typed and the allocated function has the type

$$(\forall \mathtt{X}, \mathtt{Y}.\mathit{reg}(\mathtt{X})@\mathtt{Y} \xrightarrow{\{\mathtt{X},\mathtt{Y}\}} \tau)@\mathtt{X_{sr}}$$

where τ is $\exists \mathtt{Z}.(\mathit{reg}(\mathtt{Z})@\mathtt{X}, \exists \mathtt{W}.\mathit{reg}(\mathtt{W})@\mathtt{Z})@\mathtt{X}$.

Now let us use this function. We call this function with some region handle and then delete the region t returned, which requires accessing the region handle for t, which is allocated in the region s, which is itself allocated by the function. In the code fragment below, assume that x is bound to some region handle of type $\mathit{reg}(\mathtt{X})@\mathtt{X_{sr}}$ and that u is bound to the function.

```
useregion x
in open v = (u[X, X_sr] x) as (reg(Z)@X, ∃W.reg(W)@Z)@X
   in open w = v.2 as reg(W)@Z
      in useregion v.1 in freeregion w
```

[4]We could use this argument to avoid explicit substitutions in (F2). But that may have lead to confusion in case one region variable is used to simultaneously rename more than one bound region variable.

2.4 Proof of Safety

A program is *stuck* if we reach a state (R, e) such that e is not a value and also cannot be reduced further. We show that if a program is well-typed then it does not get stuck, which implies that we can eliminate all conditions of the form $R(\mathtt{r}) = (1, -)$ in the hypotheses of the reduction rules because they only serve to make programs get stuck. Also, since no reduction turns a validity bit from 0 to 1, it is easy to see that region can be physically deleted once its validity bit goes to 0, that is, [R2] may actually free the region $\mathtt{r}$. Hence this implies that a well-typed program is memory safe.

The proof is in the style of [19]. For the proof, we must be able to type intermediate expressions. To this end, we extend the type environment so that it can map regions to region variables (e.g., $\Gamma(\mathtt{r}) = \mathtt{X}$) as well as variables to types. We omit detailed presentation of type checking rules for intermediate expressions. However, we note that `abort` has any type under any environment and that the following rule is used to type-check `inuse`'s:

$$\frac{\Gamma(\mathtt{r}) = \mathtt{X} \qquad \Gamma(\mathtt{s}) = \mathtt{Y} \qquad \Delta;\Gamma \vdash e : \tau; L \cup \{\mathtt{X}\} \qquad \mathtt{Y} \in L}{\Delta;\Gamma \vdash \mathtt{inuse}\ (\mathtt{reg}\ \mathtt{r}\ \mathtt{at}\ \mathtt{s})\ \mathtt{in}\ e : \tau; L}$$

We need the following definition for the proof.

Definition 1 *A state (R, e) is well-typed under the environment $\Delta;\Gamma$ iff*

(1) $dom(\Gamma) = dom(R)$, $ran(\Gamma) \subseteq \Delta$, and $\Delta;\Gamma \vdash e : -; \{\mathtt{X_{sr}}\}$.

(2) For each subexpression of the form $\Lambda\bullet\lambda - : \bullet.e_1$ at $-$ of e, e_1 does not contain a subexpression of the form inuse $-$ in $-$.

(3) For each $\mathtt{r} \in dom(R)$ such that $\mathtt{r} \neq \mathtt{r_{sr}}$, the number of occurrences of subexpressions of the form inuse (reg $\mathtt{r}$ at $-$) in $-$ of e is equal to i where $R(\mathtt{r}) = (-, i)$.

(4) If $R(\mathtt{r}) = (b, i)$ and $i > 0$ then $b = 1$.

We write $\Delta;\Gamma \vdash (R, e)$ to mean that (R, e) is well typed under $\Delta;\Gamma$. Given this definition, we prove the following theorem.

Theorem 1 (Subject Reduction) *If $\Delta_1;\Gamma_1 \vdash (R_1, e_1)$ and $R_1, e_1 \to R_2, e_2$, then there exists Δ_2, Γ_2 such that $\Delta_2;\Gamma_2 \vdash (R_2, e_2)$.*

Proof: The conditions (2), (3), and (4) can be proven independent of the choice of Δ_2, Γ_2. The condition (2) follows from inspection of the reduction rules. Given (2), (3) is straightforward by inspection of [R4] and [R6]. The condition (4) is straightforward by inspection of [R4].

It remains to find Δ_2 and Γ_2 to satisfy (1). We can do so by a straightforward case analysis on the redex using the substitution lemma and the replacement lemma. □ We next prove the following theorem.

Theorem 2 (Progress) *If $\Delta;\Gamma \vdash (R, e)$ and e is not a value, then there is a reduction from (R, e).*

Proof: The main part of the proof is in showing that condition of the form $R(\ldots) = \ldots$ in the hypotheses of each reduction rule is satisfied. To this end, we need the following lemma.

Lemma 1 *If $\Delta;\Gamma \vdash E[e] : -; \{\mathtt{X_{sr}}\}$ then there is a sub-derivation $\Delta;\Gamma \vdash e : -; L$ such that if $\mathtt{X} \in L$ and $\mathtt{X} \neq \mathtt{X_{sr}}$ then there is an occurrence of* `inuse (reg r at` $-$`) in` $-$ *in $E[e]$ such that $\Gamma(\mathtt{r}) = \mathtt{X}$.*

The lemma can be proven by inspection of the type checking rules. Given this lemma, we can show by (3) and (4) of Definition 1 that $R(\ldots) = \ldots$ is satisfied. □ It is easy to see that for a well-typed program e, we have $\{\mathtt{X_{sr}}\}; \{\mathtt{r_{sr}} \mapsto \mathtt{X_{sr}}\} \vdash (\{\mathtt{r_{sr}} \mapsto (1,1)\}, \mathtt{e})$ where $\mathtt{r_{sr}}$ is the starting region. So by the two theorems above, it follows that a well-typed program does not get stuck, that is, it is memory safe.

A more detailed proof for the extended version of the system appears in Appendix A.1.

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_2 : \mathit{reg}(\mathtt{X}); L_2 \qquad \mathit{fregvars}(\tau_1) \cup L_1 \subseteq \Delta \uplus \{\vec{\mathtt{Y}}\} \\ \Delta \uplus \{\vec{\mathtt{Y}}\}; \Gamma \uplus \{x \mapsto \tau_1\} \vdash e_1 : \tau_2; L_1 \qquad \mathtt{X} \in L_2\end{array}}{\Delta;\Gamma \vdash \Lambda\vec{\mathtt{Y}}\lambda\mathtt{x}{:}\tau_1.e_1 \ \mathtt{at}\ e_2 : (\forall\vec{\mathtt{Y}}.\tau_1 \xrightarrow{L_1} \tau_2)@\mathtt{X}; L_2} \text{ (F1-m)} \qquad \frac{\Delta;\Gamma \vdash \vec{e} : \vec{\tau}; L \quad \Delta;\Gamma \vdash e : \mathit{reg}(\mathtt{X}); L \quad \mathtt{X} \in L}{\Delta;\Gamma \vdash \vec{e}\ \mathtt{at}\ e : \vec{\tau}@\mathtt{X}; L} \text{ (T1-m)}$$

$$\frac{}{\Delta;\Gamma \vdash \mathtt{newregion\ at} : \exists\mathtt{X}.\mathit{reg}(\mathtt{X}); L} \text{ (R1-m)} \qquad \frac{\Delta;\Gamma \vdash e : \mathit{reg}(\mathtt{X}); L}{\Delta;\Gamma \vdash \mathtt{freeregion}\ e : \mathit{reg}(\mathtt{X}); L} \text{ (R2-m)}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \mathit{reg}(\mathtt{X}); L \quad \Delta;\Gamma \vdash e_2 : \tau; L \cup \{\mathtt{X}\}}{\Delta;\Gamma \vdash \mathtt{useregion}\ e_1\ \mathtt{in}\ e_2 : \tau; L} \text{ (R3-m)}$$

Figure 5: Modifications to (R1), (R2), and (R3)

3 Region Handles for Free

One problem with the system as described thus far is that every region handle, namely the use counter and the validity bit, needs to be allocated in a separate region from its own region (except `sr`). This problem leads to inconvenience. For example, one cannot just create a region temporarily, use it, delete it, and then get back to the same memory state because the system leaves the region handle as garbage. In order to collect this garbage, one needs to delete the region which contains this region handle, which leads back to the same problem.

While this may be a small problem in practice since a region handle takes up only a small amount of space, it may become a serious problem for severely resource limited programs. It is also annoying from the technical point of view; in almost all other manual region systems, a region handle is essentially just a pointer to the head of the region and hence requires no separate heap space (i.e., all region information is stored within the region itself).

In this section, we solve this problem at the cost of a small run-time overhead. The idea is to modify the basic system slightly so that each use counter is allocated within the region itself and the validity bit is dispensed with. Therefore a region handle no longer requires a separate space, and deleting a region does not leave a garbage region handle.

We first make a few small changes to the language to reflect the fact that region handles are self-allocated. We replace `newregion at` e with

$$\mathtt{newregion}$$

and $\mathit{reg}(\mathtt{X})@\mathtt{Y}$ with

$$\mathit{reg}(\mathtt{X})$$

We also change the intermediate representation of a region handle from `reg r at s` to

$$\mathtt{reg\ r}$$

We make a few straightforward changes to the type checking rules to reflect the changes we made. Figure 5 shows the new rules. The new rules are slightly simpler. We no longer need the starting region, so we get rid of `sr` and $\mathtt{X}_{\mathtt{sr}}$. A program e is well-typed iff $\emptyset;\emptyset \vdash e : -;\emptyset$.

Next, we introduce a new intermediate value `dr` of type $\mathit{reg}(\mathtt{X}_{\mathtt{dr}})$ where $\mathtt{X}_{\mathtt{dr}}$ is a special region variable. Note that the constant `dr` is not available to source programs.

The basic idea behind the modification is to replace each reference to a handle of a deleted region with `dr`. For each region `r`, the run-time system maintains two data-structures in `r` itself: a doubly linked list called $\mathit{reglocs}_1(\mathtt{r})$ and a singly linked list called $\mathit{reglocs}_2(\mathtt{r})$. The run-time system stores pointers to all occurrences of `reg r` in $\mathit{reglocs}_1(\mathtt{r})$. In addition, for each region `s`, $\mathit{reglocs}_2(\mathtt{r})$ stores a pointer to each occurrence of `reg s` allocated within `r`.

More concretely, at each allocation a `at reg r`, for each *direct* occurrence of `reg s` in a, the run-time system stores the address of the occurrence in $\mathit{reglocs}_1(\mathtt{s})$ and in $\mathit{reglocs}_2(\mathtt{r})$. For example, evaluating

$((\texttt{reg s}, \texttt{reg t})\ \texttt{at r}, \texttt{reg t})\ \texttt{at reg r}$ allocates an element containing the heap location of the second element of the outermost tuple (the second `reg t`) in $\mathit{reglocs}_1(\texttt{t})$ and in $\mathit{reglocs}_2(\texttt{r})$. Note that other two region handles are already allocated so nothing else needs to be done. In general, for a tuple $(v_1, \ldots, v_n)$ `at reg r`, only the v_i of the form `reg r` needs to be considered assuming that we treat `pack` $\bullet$ `as` v as v. A similar treatment is done when allocating function closures, that is, the address of any free variable bound to a region handle `reg t` gets stored in $\mathit{reglocs}_1(\texttt{t})$ and in $\mathit{reglocs}_2(\texttt{r})$ when the closure is allocated in the region `r`. Region handles allocated at non-heap locations (e.g., stack variables) are handled similarly.

A region handle is implemented as a double-word value. This implementation choice warrants further discussion later in the section (Section 3.2). A region handle `reg r` is a pair of pointers, one pointing to the head of the region `r` and the other pointing to the element of $\mathit{reglocs}_1(\texttt{r})$ corresponding to this reference.

When the program tries to delete a region `r` via some `freeregion reg r`, the run-time system loops over each element of $\mathit{reglocs}_1(\texttt{r})$ overwriting each region handle with `dr`. Then it loops over each element of $\mathit{reglocs}_2(\texttt{r})$ such that if it finds a region handle of `t` that is not `dr`, it follows the pointer to an element of $\mathit{reglocs}_1(\texttt{t})$ and deletes the element from the doubly-linked list. Finally the region `r` is deleted. Similarly, when a non-heap allocated reference to a region handle of `t` is deallocated, the run-time system follows the pointer to the corresponding element of $\mathit{reglocs}_1(\texttt{t})$ and deletes the element.

Now the use counter can be stored within the same region. We no longer need the validity bit as the run-time system just needs check for `dr` at `useregion`'s. The run-time system pays the price for the modification when allocating a value containing a reference to a region handle and when deallocating such a reference. As we shall see in Section 3.2, a suitable restriction to the static system guarantees that no other run-time cost needs to be paid. For example, the run-time system does no extra work when allocating a value that contains no immediate reference to a region handle. Deleting a region costs time proportional to the number of references to region handles allocated in that region and references to its region handle, which is independent of the number of other type of values and should be a small factor in a reasonable region program. Note that unlike with a tracing or reference-counting garbage collector, neither operation requires any reachability computation.

3.1 Mutable Values

Suppose that we extended the language with mutable values. In order to maintain the invariants just discussed, when a reference to a region handle of `r` is modified to refer to a region handle of `t`, the run-time system follows the pointer to an element of $\mathit{reglocs}_1(\texttt{r})$, deletes the element from the list, and then allocates an element in $\mathit{reglocs}_1(\texttt{t})$ containing a pointer to this reference.

3.2 Polymorphism

In order to allow smooth integration of our system with a polymorphic language, one might prefer a single-word implementation of a region handle as opposed to the double-word implementation described above. This can be accomplished by storing a pointer to the head of the region `r` in each element of $\mathit{reglocs}_1(\texttt{r})$ and implementing the region handle as a pointer to the corresponding element. This change incurs an extra-level of pointer-lookup at allocation sites.

But there is a strong reason not to allow instantiating a type variable with a region handle type in the presence of polymorphism. The problem is that, at allocation sites, the run-time system needs to check if a value whose type is some type variable (at compile time) is a region handle, and if so the corresponding linked lists need to be extended. For example, when allocating a value of the type $(\alpha, \mathit{reg}(\texttt{Y}))@\texttt{X}$ where α is a type variable, the run-time system needs to test if the first element of the tuple is a region handle. A similar run-time overhead occurs when updating a mutable value that has a type-variable type.

In contrast, if we forbid abstraction over a region handle type, then all references to region handles will be known at compile time. Note that programs are still allowed to instantiate a "boxed" type that contains a region handle type (e.g., a heap-allocated tuple containing a region handle: $\tau[(\mathit{reg}(\texttt{X}), \mathit{reg}(\texttt{Y}))@\texttt{Z}/\alpha]$). This design also lets us keep the double-word region handle implementation. Because unboxed multi-word values appear to be important for efficient manual memory management in a full-scale programming language [9], having a region handle type as just another non-generalizable type might not cause further damage to the overall uniformity of the language.

It is worth noting that, in case one wants abstraction over region handle types despite its shortcomings, it is possible to alleviate the run-time overhead via static analysis. For example, a flow analysis can conservatively estimate which value may not evaluate to a region handle. We leave the choice open to the language designer.

4 Analysis and Discussion

In this section, we evaluate our use-counting technique analytically and discuss several pragmatic issues. First, we show an efficient encoding of stack-of-regions style memory management within our system to show that use counting is at least as powerful as this well-known region technique.

4.1 Encoding Stack-of-Regions

Stack-of-regions refers to a region-based memory management technique where each region's lifetime is bounded by the lifetime of some stack frame. This technique is the basis of many region proposals, both manual and automatic [15, 1, 9, 2].

In this paper, we are concerned with the manual interpretation. The basic manual stack-of-regions can be obtained by removing `newregion`, `freeregion`, and `useregion` from our small region language and adding

$$\texttt{letregion}\ \texttt{x}\ \texttt{as}\ \mathit{reg}(\texttt{X})\ \texttt{in}\ e$$

Reducing this expression creates a new region referred to by `X`, substitutes the region handle of the new region for `x` in e, reduces e to a value v, deletes the region, and then returns v.

The type checking rule is

$$\frac{\begin{array}{c}\Delta \uplus \{\texttt{X}\}; \Gamma \uplus \{\texttt{x} \mapsto \mathit{reg}(\texttt{X})\} \vdash \texttt{e} : \tau; \texttt{L} \cup \{\texttt{X}\} \\ \mathit{fregvars}(\tau) \subseteq \Delta\end{array}}{\Delta; \Gamma \vdash \texttt{letregion}\ \texttt{x}\ \texttt{as}\ \mathit{reg}(\texttt{X})\ \texttt{in}\ e; \tau; L}$$

We show that `letregion` can be efficiently encoded using `newregion`, `freeregion`, and `useregion`. Let $e_1; e_2$ be an abbreviation for `let x =` e_1 `in` e_2 where $\texttt{x} \notin \mathit{fvars}(e_2)$. Now consider:

$$\begin{array}{l}\texttt{open x = (newregion) as } \mathit{reg}(\texttt{X}) \\ \texttt{in let y = (useregion x in } e\texttt{) in freeregion x; y}\end{array}$$

It is easy to see that this encoding has the same behavior as `letregion`. That is, before entering e, a new region is created and bound to `x`, then after e reduces to a value, the region is deleted. It is also easy to see that the encoding preserves well-typedness.

The resulting program is efficient. There is only a constant time overhead for each occurrence of `letregion`. That is, before evaluating the body, a new region and its use counter is allocated, and the use counter is incremented after checking that the region has not been deleted. Once the body evaluates, the use counter is decremented, and the region is deleted after checking that the use counter is 0.

It is easy to see that a translated stack-of-regions program will never `abort`.

4.2 Dynamic Check Failures

However, the full potential of use counting is realized when the program does not use regions in a stack-wise order. For example, stack-of-regions is insufficient to express the example in Section 2.3 since every function is forced to delete all regions it creates before it returns.

In the general case, however, programs do not enjoy the `abort`-free property. But we argue that it is rare that a well-written program encounters a dynamic check failure. Recall that there are two kinds of dynamic checks: the `dr` check (or the validity bit check) at `useregion` and the use counter check at `freeregion`. Consider a situation in which the first check fails, that is, `useregion reg r in` e is encountered after the region `r` has been deleted. Suppose that the programmer adheres to the discipline that, by `useregion reg r in` e, he/she means that `r` is definitely going to be used in e. Then a dynamic check failure indicates a real programming bug instead of a spurious dynamic check failure since `r` was deleted before executing the code which was supposed to use `r`.

On the other hand, the second dynamic check fails when `freeregion reg r` is encountered within some context `inuse reg r in` E (at least for the single-thread case). This type of failure is harder to characterize. But it is worth noting that conservatively detecting such failures at compile time is possible to some degree via known static analysis techniques. For example, a variation of the algorithm from [5] could be used to produce compile-time warnings for potentially dangerous `freeregion` occurrences. Also, this type of failure is easier to recover from, since the worst the programmer could do is to ignore and leak memory.

4.3 Annotation Cost

To use the system, the programmer needs to annotate code with `useregion` constructs in addition to writing type and effect annotations. Somewhat surprisingly, `useregion` and type and effect annotations are mutually compensating. Intuitively, the more `useregion` annotations the programmer writes, the fewer type and effect annotations required.

To illustrate this point, consider a large block of code e_0 that uses many regions, say `r, s, ...`. Assume that e_0 does not delete these regions. Then it is a good idea to wrap e_0 with corresponding `useregion`'s to amortize the cost of dynamic checks since e_0 is large. That is,

```
useregion $e_1$
in useregion $e_2$
   in useregion ... in $e_0$
```

where e_1, e_2, ... are region handles for `r, s, ...`, respectively. Now consider a function which contains e_0 as a subexpression, say, $\lambda x : \tau_1 . e_0$ `at` $-$ for the sake of brevity. Then, the type of this function is just $(\tau_1 \xrightarrow{\emptyset} \tau_2)$@`X` for some τ_2 and `X` assuming that e_0 uses no other regions.

In general, an occurrence of `useregion` eliminates the corresponding region variable from the effect sets of the parent expressions, which in turn relieves the programmer from having to annotate them in the corresponding effect sets. We can see a glimpse of this property in action in the examples in Section 6 where every function has a rather trivial effect set annotation. This property also has an compounding benefit because a context calling such function requires neither `useregion` nor effect annotations for the regions accessed by the callee. This is in contrast to conventional region-based approaches where function types are required to carry identifications of all non-local regions accessed by the function (up to abstraction over sets of region identifiers).

On the other hand, for run-time efficiency, it is desirable to leave the body of a short-running function free of `useregion`'s. Such a small function is likely to use few regions, and therefore the effort required to annotate its effect set is correspondingly small.

The system presented in this paper is explicitly typed. While outside of the scope of the paper, it is easy to see that inferring type and effect annotations is possible with some suitable restrictions. Some amount of inference (e.g., in the spirit of local type inference [13]) will be important for incorporating use counting in a full-scale programming language.

5 Regions and Threads

A problem common with many manual memory management systems is that they have difficulty supporting multi-threading while retaining safety and controllability. Consider the stack-of-regions approach. To handle thread-shared memory, the run-time system must be aware of threads that may potentially access a region so that the region may be deleted only after all potentially accessing threads terminate. If the thread that allocated the region is allowed to continue after the stack pops regardless of future behavior of other threads, then memory behavior becomes difficult to control because the lifetime of the region is no longer tied to the lifetime of the stack but to the longest living thread having access to the region. On the other hand, if the thread that allocated the region is forced to stall until it is safe to delete, then the run-time behavior of the allocating thread becomes less predictable.

Thread-based concurrency is common in many modern programs. One of the nicest features of the use-counting approach is that thread-based concurrency is supported for free.

We extend the small region language with the form `fork` e and extend the dynamic semantics with concurrently executing threads. Each state has the form $(R, \vec{e})$ where each $e_i \in \vec{e}$ is a thread. Reductions are still from states to states, and each reduction rule can be applied to any thread in $\vec{e}$. The reduction rule for `fork` e is

$$R, (\vec{e_1}, E[\texttt{fork}\, e], \vec{e_2}) \to R, (\vec{e_1}, E[\texttt{unit}], \vec{e_2}, e)$$

where the constant `unit` is a place holder value of type *unit*. The type checking rule for `fork` e is

$$\frac{\Delta; \Gamma \vdash e : \tau; \emptyset}{\Delta; \Gamma \vdash \texttt{fork}\, e : \mathit{unit}; L}$$

No other changes are needed.

The argument used in the proof of safety for the sequential case still works. The key observation is that the proof of safety was based on the number of syntactic occurrences of `inuse`s. Hence to prove safety for multi-thread case, we can simply add up the number of `inuse`s from all threads and compare the sum with the use counter in the region map. Since $\Delta; \Gamma \vdash e : \tau; \emptyset$ and since e does not contain any `inuse`, the newly forked thread e satisfies the invariants in Definition 1, and therefore the induction argument goes through. Appendix A.1 discusses the reasoning in more detail.

As in the single-thread case, deleting a region while being used or trying to use a region that has been deleted results in a dynamic check failure. The difference is that such situations may now involve concurrently running threads, for example, when thread A attempts to delete some region that thread B is using. To avoid dynamic check failures, programs must synchronize threads so that region uses and deletions are correctly scheduled.

How such synchronization is done, whether explicitly by the programmer or implicitly by concurrency abstractions in the programming language, is another issue. Our proposal provides a simple and sound basis for incorporating safe memory management with any synchronization mechanism.

6 Experience with Regions

We informally discuss our experience with a toy implementation of the technique. Instead of creating an entirely new language from scratch, we extend an existing language with use-counted regions. We choose C for this purpose, mainly because we had prior experience with manual, though sometimes unsafe, region programming in this language.

The ideal goal is to replace `malloc` and `free` with `newregion` and `freeregion`. But more realistically, our implementation provides use-counted regions as a safe alternative while keeping the rest of C available and unsafe. Hence we leave features such as casting in the language. The implementation covers only the monomorphic fragment of the system. (Monomorphic as in the sense of Section 2, and so we do permit quantification over region variables.) However, we support non-heap-allocated records in the form of `struct`s, which turns out to be useful for writing short but interesting examples. One reason for this section is to provide ample examples in easy-to-read syntax. The examples are designed to illustrate the expressive power of our system such that other safe, explicit memory management approaches would have difficulty expressing them.

6.1 Syntax

The basic scheme is identical to that of the small region language. The built-in function `region newregion(void)` creates a new region and returns its region handle, and the built-in function `void freeregion(region)` deletes the region passed. We also have `useregion`, which is now written `useregion(`e_1`) {` e_2 `}`.

However, C has a number of differences with the lambda calculus (to put it mildly) which dictate a somewhat different design. The major changes are:

1 Instead of the "@" notation, we now indicate where a value is stored as part of its pointer type. For example, `int * X` is a pointer to an integer in region referred to by `X`. We allow `NULL` to have any pointer type.

2 We use syntax resembling C++ templates to write data structure types parametrized by region variables.

$$\texttt{struct<}\vec{\texttt{X}}\texttt{>}\mathit{structname}\{\ \mathit{def}\ \}$$

The region variables $\vec{\texttt{X}}$ are binding occurrences, and therefore $\vec{\texttt{X}}$ may appear free in *def*. For such a type τ, its region variables can be instantiated: τ`<`$\vec{\texttt{X}}$`>`. What is unique about our `struct`'s is that when it is not instantiated, the value is said to be *closed*, which intuitively represents an existential type abstracted over the bound region variables. For example, a newly created region handle has the type `region` and is therefore closed.

For example, the type `IntCell` consisting of a region handle `reg` to some region `r` and a pointer `data` pointing to an integer allocated in `r` is written as

```
struct<Z> IntCell { region<Z> reg; int * Z data };
```

An instantiated type can be closed at any time, which intuitively represents `pack`.

3 The built-in polymorphic function

$$\tau\ \texttt{* X ralloc(region<X>,}\tau\texttt{) \{X\}}$$

allocates values of type τ in the region passed and returns a pointer to the allocated memory block. Allocated memory blocks are initialized to zero. The set `{X}` denotes the latent effect of the function. Latent effect annotations are omitted for functions whose latent effect is $\emptyset$.

4 We use C conventions of compound statement scope. For example, to open e of closed type τ by instantiating it with region variables `X,Y,Z`, write

$$\texttt{open}\ \tau\texttt{<X,Y,Z> x = }e\texttt{;}$$

Then `X,Y,Z` are new region variables that are available in the current scope. The variable `x` is forced to be `const`.

5 We drop explicit polymorphic functions. Instead, every function is always implicitly polymorphic in free region variables appearing in its return type, argument types, and latent effect because C functions are always defined at the top level. Functions are instantiated at the call-sites.

```
struct elt {
   struct<X> wrapper {
      region<X> reg;
      struct elt * X next;
   } wrap;
   int data;
};
void insert(struct elt * X cur, int newdata) {X} {
   open region<Y> newreg = newregion();
   struct elt * Y newelt;
   useregion(newreg) {
      newelt = ralloc(newreg, struct elt);
      newelt->data = newdata;
      newelt->wrap = cur->wrap;
   }
   cur->wrap = {reg:newreg, next:newelt};
}
void delete(struct elt * X cur) {X} {
   open struct wrapper<Y> cw = cur->wrap;
   useregion(cw.reg) {
      cur->wrap = cw.next->wrap;
   }
   freeregion(cw.reg);
}
void print_list(struct wrapper list) {
   do { open struct wrapper<X> cw = list;
         if (cw.next == NULL) return;
         useregion(cw.reg) {
            printf(‘%d‘,cw.next->data);
            list = cw.next->wrap;
         } } while (1);
}
```

Figure 6: Element-wise deallocatable linked list

6.2 Example 2: Linked List

Element-wise deallocatable linked lists can be implemented in our system with full imperative, destructive goodness expected from lists in C. To this end, we use one region per list element; the definition of `elt` in Figure 6 shows that a list element consists of a region handle, a next pointer, and a content data. The content data is just an integer, so the example is probably not an economical use of regions: it is just for demonstration.

Figure 6 shows three functions that do insertion, delete, and list printing, respectively.[5] Note that deleting a list element actually deallocates the space for the element while keeping the rest of the list intact. All functions are written in an imperative fashion common in C programs; there is no need to rebuild the list when an element is inserted or deleted.

Now suppose instead of an integer as a list element, we want a large data structure potentially consisting of many connecting pointers. Suppose that this data structure is to be allocated in the same region where

[5]The last statement in `insert` is not valid C syntax because `...= {...}` can be used only for initializations. This can be overcome by either extending C or adding an extra scope at the end initializing a new variable of type `elt` which gets immediately assigned to `cur->wrap`.

```
struct<X> gen {
   region<X> reg;
   struct game<X> * X g;
};
struct gen nextgen(struct gen);
struct gen life(int n, struct gen g) {
   if (n == 0) { return g; }
   else {
      struct gen newg = nextgen(g);
      open struct gen<X> x = g; freeregion(x.reg);
      return life(n-1, newg);
   }
}
```

Figure 7: Game of Life simulation

the element is allocated (at the element creation or later). To this end, we need a region variable referring to the region where the element is allocated. We add a new binding region variable in the definition of `elt` and using this binding to equate the region where this instance of `elt` is allocated and the region where the data structure is (or is going to be) allocated.

The updated definition of a list element appears below. The `struct` type `data_type` (the definition not shown) is instantiated with `Y` for pointers to values that consist this data structure.

```
struct<Y> elt {
   struct<X> wrapper {
      region<X> reg;
      struct elt<X> * X next;
   } wrap;
   struct data_type<Y> * Y data;
};
```

6.3 Example 3: Game of Life

An example simulating the Game of Life for n iterations is used in [10] to illustrate varying expressiveness of different region systems. In Figure 7, we show a use counting implementation which closely matches the original code found in their paper.[6] The function `nextgen` creates and returns the new generation.

The resulting function `life` satisfies both of the two criteria suggested by their paper: it is tail recursive and does not have the *region endomorphism* problem (i.e., using the same region to allocate the generations, which leads to memory leak).

6.4 Existing Region Programs

We also looked at several existing C programs that use regions (safely or not) to do part of their memory management and found that use counting seems capable of expressing many region usage patterns found in these programs.

One common pattern found in several programs is that there are a fixed number of regions existing at any time, and these regions are stored in global variables (or accessed by calling global functions). For example, `lcc`, a C compiler, uses three regions placed in a global array of size three called `arena`: one for functions, one for statements, and one permanent region. The region deletion function `deallocate(int i)` is called

[6][10] also allocates the integer argument `n` in the heap. Here we just stack-allocate it for brevity.

to delete the current `arena[i]` after each function (or each statement) is compiled. Each `arena[i]` (except of course the permanent one) is assigned a newly created region before the next allocation in `arena[i]`.

It is easy to express such a pattern in our system. We store the region handle with the data in a globally accessible closed `struct` and open it wherever the data is requested. We reset the region by deleting the region, creating a new one and closing it in a `struct`, and assigning the closed value to the same global variable.

Because most of these programs are neither written to use regions for safety nor pressured with tight space or timing requirements, they do not contain too many exciting region usage patterns. But it gives us some practical evidence for the technique's usability.

7 Related Work

Ruggeieri et al. [14] introduced the stack-of-regions concept, and Tofte et al. [15] extended it to work with higher-order functions and polymorphic regions. Both systems were formulated for automatic memory management, but manual variants of the idea have also been proposed [9, 2]. We have given a detailed comparison of use-counted regions versus stack-of-regions in the earlier parts of the paper. We should note that issues that are not formally discussed in the main body of our paper, such as parametric polymorphism over types and effect sets, recursive types, non-heap allocated values, and mutable values, have been extensively studied in the aforementioned papers. It is straightforward to incorporate these features formally into our system in much the same way, due partly to the fact that we use a similar framework (with the minor exception of the issue discussed in Section 3.2). Appendix A shows the small region language extended with ML-style mutable refs and parametric polymorphism over types and effect sets.

Hicks et al. have developed a similar technique called *dynamic regions* independently from our work around the same time. In [11], they briefly outline this technique along with two other memory management ideas that are of independent interest. Their dynamic regions can be roughly interpreted in our small region language by assuming that region handles are always existentially packed. Opening such package in their system corresponds to `open` immediately followed by `useregion`. They have incorporated dynamic regions in Cyclone [9] and report positive results. Compared to their work, our paper offers the following contributions: a formal presentation of the technique and the proof of soundness, analysis and discussion including the exposition on multi-threading support, and the following two technical differences regarding region handles. As mentioned, Hicks et al.'s formulation ties dynamic operations on region handles to existential types, whereas we use an arguably more natural formulation which treats existential types as purely a static construct as in the standard type theory. In addition to being somewhat cleaner in theory, our formulation has practical advantages. For example, our system allows two different existential abstractions over the same region handle, which in turn, for example, enables sharing of a region among different data structures without having to syntactically equate the region variables. The second difference is the extension shown in Section 3 which allows a region handle to be allocated within its own region.

Walker et al. [18] combine regions with linear types to create a static system that allows interesting region usage patterns. In their system, each region handle must be linear when the region is deleted and non-linear (and therefore allowed to be aliased) when the region needs to be accessed. Wadler's `let!`-like construct [16] turns a linear region handle non-linear in a designated scope and then back to linear when exiting the scope. Walker et al.'s system uses these linear/non-linear phases to statically prevent a region from being deleted while program is in one of its non-linear scopes. Hence the underlying idea is somewhat similar to our `useregion`, and so our use-counting method may be viewed as a way to remove limitations imposed by linear types by inserting dynamic checks. Their paper also shows how to integrate a reference-counting interpretation of linear types [3] with regions.

Crary et al. [4] present a region system that uses the notion of linearity to statically track aliasing of region handles. Their system is very expressive but seems to require non-trivial program annotations, for example, when deallocating individual elements of recursive data structures ala our example in Section 6.2 [17]. However, their intention is not to expose the region system at the source level but instead to certify memory safety of intermediate-level code.

Henglein et al. [10] takes a different approach toward region-based memory management by employing a Floyd-Hoare style proof system. Their primary goal is to create an automatic system for a conventional non-region language ala [15, 1]. Hence their system is not meant to be exposed to programmers and also not quite as manual as the systems discussed above. In particular, there is no language construct to explicitly delete a region. Instead, the system maintains reference counts on region handles[7] so that a region is automatically deleted when the reference count becomes zero. To this end, they use a reference-counting scheme similar in spirit to the one in [3]. It is left open whether their system could be extended to handle function closures and polymorphism.

In contrast, Gay et al. [6] uses reference counting as the main device for memory safety by counting references to the content of regions in addition to region handles. Their system allows explicit deletion of regions but does not statically guarantee its success, similar to our `freeregion`. Their system demands very little from the static side, relying largely on dynamic reference counting for safety. To recover efficiency, they use compile-time optimization to reduce the number of reference-counting operations and also provide optional type qualifier annotations to help the optimizing compiler.

This reference-counting approach is related to use counting in the following way. Both approaches count the "users" of each memory location so that the memory location may be deallocated only when the number of users is zero. For efficiency, both frameworks group memory locations via regions to reduce the number of counters. The difference between the two approaches is in the meaning of a "user": in the reference-counting approach, a user is a pointer pointing to the region whereas in our approach, a user is a piece of code that is actually *using* the region as in accessing memory locations in the region. Hence to group memory locations, the reference-counting approach prefers connecting pointers to be in the same group, whereas our approach encourages grouping memory locations that are accessed together in the same piece of code. This observation has led us to coin the term "use counting" to describe our technique.

Our use-counting approach has the following advantages over the reference-counting approach. One is that we do not require the references to be dead for a region to be deleted. Another advantage is that, when deciding which locations should be grouped together in a region, use counting can capitalize directly on the program's memory access pattern instead of being dependent on the points-to relationship.

8 Conclusion

We have presented a new, mostly-static, safe region-based memory management technique. Our method scales naturally to a variety of modern language features, including multi-threading, and offers flexible manual control while being relatively simple. We hope to have provided a new insight into designing high-level languages for resource-conscious applications.

References

[1] A. Aiken, M. Fähndrich, and R. Levien. Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, La Jolla, California, June 1995.

[2] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, California, June 2003.

[3] J. Chirimar, C. A. Gunter, and J. G. Riecke. Reference Counting as a Computational Interpretation of Linear Logic. *Journal of Functional Programming.*

[7] Region variables in their paper.

[4] K. Crary, D. Walker, and G. Morrisett. Typed Memory Management in a Calculus of Capabilities. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, Texas, Jan. 1999.

[5] J. S. Foster and A. Aiken. Checking Programmer-Specified Non-Aliasing. Technical Report UCB//CSD-01-1160, University of California, Berkeley, Oct. 2001.

[6] D. Gay and A. Aiken. Language Support for Regions. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, Snowbird, Utah, June 2001.

[7] D. K. Gifford, P. Jouvelot, J. M. Lucassen, and M. A. Sheldon. FX-87 Reference Manual. Technical Report MIT/LCS/TR-407, MIT Laboratory for Computer Science, Sept. 1987.

[8] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-Based Memory Management in Cyclone. Technical Report 2001, Department of Computer Science, Cornell University, 2001.

[9] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-Based Memory Management in Cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[10] F. Henglein, H. Makholm, and H. Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd International Conference on Principles and Practice of Declarative Programming*, pages 175–186, Montréal, Canada, 2001. ACM.

[11] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Safe and Flexible Memory Management in Cyclone. Technical Report CS-TR-4514, Department of Computer Science, University of Maryland, July 2003.

[12] J. M. Lucassen and D. K. Gifford. Polymorphic Effect Systems. In *Proceedings of the 15th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–57, San Diego, California, Jan. 1988.

[13] B. C. Pierce and D. N. Turner. Local Type Inference. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 252–265, San Diego, California, Jan. 1998.

[14] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings of the 15th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 285–293, San Diego, California, Jan. 1988.

[15] M. Tofte and J.-P. Talpin. Implementation of the Typed Call-by-Value λ-Calculus using a Stack of Regions. In *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, Portland, Oregon, Jan. 1994.

[16] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.

[17] D. Walker and G. Morrisett. Alias Types for Recursive Data Structures. In *International Workshop on Types in Compilation*, Montreal, Canada, Sept. 2000.

[18] D. Walker and K. Watkins. On Regions and Linear Types. In *Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming*, pages 181–192, Florence, Italy, Sept. 2001.

[19] A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.

$$\mathtt{U},\mathtt{V},\mathtt{W},\mathtt{X},\mathtt{Y},\mathtt{Z} \in \text{RegVars} \cup \{\mathtt{X_{sr}}\} \quad \mathtt{u},\mathtt{v},\mathtt{w},\mathtt{x},\mathtt{y},\mathtt{z} \in \text{Vars} \cup \{\mathtt{sr}\} \quad \mathtt{r},\mathtt{s},\mathtt{t} \in \text{Regions}$$
$$\mathtt{a},\mathtt{b},\mathtt{c} \in \text{TypeVars} \quad \mathtt{A},\mathtt{B},\mathtt{C} \in \text{EffectVars} \quad \mathtt{k},\mathtt{l},\mathtt{m} \in \text{Locations}$$

$$\begin{array}{llll}
\text{static vars} & \alpha & ::= & \mathtt{a} \mid \mathtt{A} \mid \mathtt{X} \\
\text{instantiations} & \gamma & ::= & \tau \mid L \mid \mathtt{X} \\
\text{effect sets} & L & ::= & \emptyset \mid L \cup \{X\} \mid L \cup \mathtt{A} \\
\text{types} & \tau & ::= & (\forall\vec{\alpha}.\tau_1 \xrightarrow{L} \tau_2)@\mathtt{Y} \mid \vec{\tau}@\mathtt{X} \mid \mathit{reg}(\mathtt{X})@\mathtt{Y} \mid \exists\alpha.\tau \mid \mathit{unit} \mid \mathtt{A} \mid \mathit{ref}(\tau)@\mathtt{X} \\
\text{expressions} & e & ::= & \mathtt{x} \mid \mathtt{let\ x = } e_1 \mathtt{\ in\ } e_2 \mid \Lambda\vec{\alpha}\lambda\mathtt{x}{:}\tau.e_1 \mathtt{\ at\ } e_2 \mid e_1[\vec{\gamma}]\ e_2 \mid \vec{e} \mathtt{\ at\ } e \mid e.i \mid \mathtt{pack\ } e \mathtt{\ as\ } \exists\alpha.\tau \mid \mathtt{open\ x = } e_1 \mathtt{\ as\ } \tau \mathtt{\ in\ } e_2 \mid \\
 & & & \mathtt{newregion\ at\ } e \mid \mathtt{freeregion\ } e \mid \mathtt{useregion\ } e_1 \mathtt{\ in\ } e_2 \mid \mathtt{fork\ } e \mid \mathtt{ref\ } e_1 \mathtt{\ at\ } e_2 \mid \mathtt{!}e \mid e_1 \mathtt{\ := } e_2
\end{array}$$

Figure 8: Extended small region language: source syntax

$$\begin{array}{llll}
\text{values} & v & ::= & \mathtt{pack\ } v \mathtt{\ as\ } \bullet \mid \Lambda\bullet\lambda\mathtt{x}{:}\bullet.e \mathtt{\ at\ r} \mid \vec{v} \mathtt{\ at\ r} \mid \mathtt{reg\ r\ at\ s} \mid \mathtt{abort} \mid \mathtt{unit} \mid \mathtt{loc\ l\ at\ r} \\
\text{expressions} & e & ::= & \mathtt{x} \mid \mathtt{let\ x = } e_1 \mathtt{\ in\ } e_2 \mid \Lambda\bullet\lambda\mathtt{x}{:}\bullet.e_1 \mathtt{\ at\ } e_2 \mid e_1[\bullet]\ e_2 \mid \vec{e} \mathtt{\ at\ } e \mid e.i \mid \mathtt{pack\ } e \mathtt{\ as\ } \bullet \mid \mathtt{open\ x = } e_1 \mathtt{\ as\ } \bullet \mathtt{\ in\ } e_2 \mid \\
 & & & \mathtt{newregion\ at\ } e \mid \mathtt{freeregion\ } e \mid \mathtt{useregion\ } e_1 \mathtt{\ in\ } e_2 \mid v \mid \mathtt{inuse\ (reg\ r\ at\ s)\ in\ } e \mid \\
 & & & \mathtt{fork\ } e \mid \mathtt{ref\ } e_1 \mathtt{\ at\ } e_2 \mid \mathtt{!}e \mid e_1 \mathtt{\ := } e_2
\end{array}$$

$$\begin{array}{llll}
\text{contexts} & E & ::= & [\,] \mid \mathtt{let\ x = } E \mathtt{\ in\ } e \mid \Lambda\bullet\lambda\mathtt{x}{:}\bullet.e \mathtt{\ at\ } E \mid E[\bullet]\ e \mid v[\bullet]\ E \mid \\
 & & & (\vec{v}, E, \vec{e}) \mathtt{\ at\ } e \mid \vec{v} \mathtt{\ at\ } E \mid \mathtt{pack\ } E \mathtt{\ as\ } \bullet \mid \mathtt{open\ x = } E \mathtt{\ as\ } \bullet \mathtt{\ in\ } e \mid \\
 & & & \mathtt{newregion\ at\ } E \mid \mathtt{freeregion\ } E \mid \mathtt{useregion\ } E \mathtt{\ in\ } e \mid \mathtt{inuse\ (reg\ r\ at\ s)\ in\ } E \mid \\
 & & & \mathtt{ref\ } E \mathtt{\ at\ } e \mid \mathtt{ref\ } v \mathtt{\ at\ } E \mid \mathtt{!}E \mid E \mathtt{\ := } e \mid (\mathtt{loc\ l\ at\ r}) \mathtt{\ := } E
\end{array}$$

Figure 9: Extended small region language: type-erased intermediate expressions and evaluation contexts

A Imperative, Concurrent and Polymorphic Small Region Language

Figure 8 shows the small region language extended with quantification over type and effect variables and ML-style mutable refs. Recursive functions can be easily expressed using refs. We also formally add the multi-thread extension discussed in Section 5. Figure 9 shows the corresponding type-erased intermediate expressions and evaluation contexts. For the purpose of exposition, we stick with the notation from Section 2 and do not use the extension described in Section 3.

The dynamic semantics is a series of small-step reductions from states to states where state is a triple $(S, R, \vec{e})$ where S is a *store* mapping each location $\mathtt{l}$ in its domain to a value. For brevity, we define the multi-thread-evaluation context:

$$T ::= (\vec{e_1}, E, \vec{e_2})$$

Figure 10 shows the corresponding reductions rules.

Figure 11 shows the type checking rules for the source language. Δ's may contain type variables and effect variables in addition to region variables. The set $\mathit{fvars}(\vec{\gamma})$ denotes the set of free static variables (i.e., region variables, type variables, and effect variables) in $\vec{\gamma}$. The bound static variables that appear in a function type are assumed to be distinct. We assume that substitution (F2-e) matches not only the number of static variables against the instantiations but also their kinds, and similarly for (E1-e) (i.e., each type variable is substituted by a type, each region variable is substituted by a region variable, and each effect variable is substituted by an effect set). The relation $L_1 \subseteq L_2$ is defined as follows:

$$\frac{}{\mathtt{A} \subseteq \mathtt{A}} \qquad \frac{}{\emptyset \subseteq \mathtt{A}} \qquad \frac{L_1 \subseteq L_2 \quad L_3 \subseteq L_4}{L_1 \cup L_3 \subseteq L_2 \cup L_4}$$

in addition to the usual axioms for the set theoretic interpretation of $\subseteq$. We also assume that $\mathtt{A} \cup \mathtt{A} = \mathtt{A}$ and that $\cup$ is commutative and associative.

$$\frac{R(\mathtt{r}) = (1,-) \qquad R(\mathtt{s}) = (1,-)}{S,R,T[\Lambda\bullet\lambda\mathtt{x}{:}\bullet.e\ \mathtt{at}\ (\mathtt{reg}\ \mathtt{r}\ \mathtt{at}\ \mathtt{s})] \to S,R,T[\Lambda\bullet\lambda\mathtt{x}{:}\bullet.e\ \mathtt{at}\ \mathtt{r}]}\ \text{[F1-e]} \qquad \frac{R(\mathtt{r}) = (1,-)}{S,R,T[(\Lambda\bullet\lambda\mathtt{x}{:}\bullet.e\ \mathtt{at}\ \mathtt{r})[\bullet]\ v] \to S,R,T[e[v/\mathtt{x}]]}\ \text{[F2-e]}$$

$$\frac{}{S,R,T[\mathtt{open}\ \mathtt{x}\ \mathtt{=}\ (\mathtt{pack}\ v\ \mathtt{as}\ \bullet)\ \mathtt{as}\ \bullet\ \mathtt{in}\ e] \to S,R,T[e[v/\mathtt{x}]]}\ \text{[E-e]} \qquad \frac{R(\mathtt{r}) = (1,-) \qquad R(\mathtt{s}) = (1,-)}{\begin{array}{l}S,R,T[\mathtt{newregion}\ \mathtt{at}\ (\mathtt{reg}\ \mathtt{r}\ \mathtt{at}\ \mathtt{s})] \\ \quad \to S,R \uplus \{\mathtt{t} \mapsto (1,0)\},T[\mathtt{pack}\ (\mathtt{reg}\ \mathtt{t}\ \mathtt{at}\ \mathtt{r})\ \mathtt{as}\ \bullet]\end{array}}\ \text{[R1-e]}$$

$$\frac{R(\mathtt{s}) = (1,-)}{\begin{array}{l}S,R \uplus \{\mathtt{r} \mapsto (-,0)\},T[\mathtt{freeregion}\ (\mathtt{reg}\ \mathtt{r}\ \mathtt{at}\ \mathtt{s})] \\ \qquad \to S,R \uplus \{\mathtt{r} \mapsto (0,0)\},T[\mathtt{reg}\ \mathtt{r}\ \mathtt{at}\ \mathtt{s}]\end{array}}\ \text{[R2-e]} \qquad \frac{R(\mathtt{s}) = (1,-) \qquad i > 0}{\begin{array}{l}S,R \uplus \{\mathtt{r} \mapsto (-,i)\},T[\mathtt{freeregion}\ (\mathtt{reg}\ \mathtt{r}\ \mathtt{at}\ \mathtt{s})] \\ \qquad \to S,R \uplus \{\mathtt{r} \mapsto (-,i)\},\mathtt{abort}\end{array}}\ \text{[R3-e]}$$

$$\frac{R(\mathtt{s}) = (1,-)}{S,R \uplus \{\mathtt{r} \mapsto (1,i)\},T[\mathtt{useregion}\ (\mathtt{reg}\ \mathtt{r}\ \mathtt{at}\ \mathtt{s})\ \mathtt{in}\ e] \to R \uplus \{\mathtt{r} \mapsto (1,i+1)\},T[\mathtt{inuse}\ (\mathtt{reg}\ \mathtt{r}\ \mathtt{at}\ \mathtt{s})\ \mathtt{in}\ e]}\ \text{[R4-e]}$$

$$\frac{R(\mathtt{s}) = (1,-)}{\begin{array}{l}S,R \uplus \{\mathtt{r} \mapsto (0,i)\},T[\mathtt{useregion}\ (\mathtt{reg}\ \mathtt{r}\ \mathtt{at}\ \mathtt{s})\ \mathtt{in}\ e_2] \\ \qquad \to S,R \uplus \{\mathtt{r} \mapsto (0,i)\},\mathtt{abort}\end{array}}\ \text{[R5-e]} \qquad \frac{R(\mathtt{s}) = (1,-)}{\begin{array}{l}S,R \uplus \{\mathtt{r} \mapsto (b,i)\},T[\mathtt{inuse}\ (\mathtt{reg}\ \mathtt{r}\ \mathtt{at}\ \mathtt{s})\ \mathtt{in}\ v] \\ \qquad \to S,R \uplus \{\mathtt{r} \mapsto (b,i-1)\},T[v]\end{array}}\ \text{[R6-e]}$$

$$\frac{R(\mathtt{r}) = (1,-) \qquad R(\mathtt{s}) = (1,-)}{S,R,T[\vec{v}\ \mathtt{at}\ (\mathtt{reg}\ \mathtt{r}\ \mathtt{at}\ \mathtt{s})] \to S,R,T[\vec{v}\ \mathtt{at}\ \mathtt{r}]}\ \text{[T1-e]} \qquad \frac{R(\mathtt{r}) = (1,-)}{S,R,T[(\vec{v}\ \mathtt{at}\ \mathtt{r}).i] \to S,R,T[v_i]}\ \text{[T2-e]}$$

$$\frac{}{S,R,T[\mathtt{let}\ \mathtt{x}\ \mathtt{=}\ v\ \mathtt{in}\ e] \to S,R,T[e[v/x]]}\ \text{[L-e]}$$

$$\frac{R(\mathtt{r}) = (1,-) \qquad R(\mathtt{s}) = (1,-)}{S,R,T[\mathtt{ref}\ v\ \mathtt{at}\ (\mathtt{reg}\ \mathtt{r}\ \mathtt{at}\ \mathtt{s})] \to S \uplus \{\mathtt{l} \mapsto v\},R,T[\mathtt{loc}\ \mathtt{l}\ \mathtt{at}\ \mathtt{r}]}\ \text{[M1-e]} \qquad \frac{R(\mathtt{r}) = (1,-)}{\begin{array}{l}S \uplus \{\mathtt{l} \mapsto v\},R,T[\mathtt{!}(\mathtt{loc}\ \mathtt{l}\ \mathtt{at}\ \mathtt{r})] \\ \qquad \to S \uplus \{\mathtt{l} \mapsto v\},R,T[v]\end{array}}\ \text{[M2-e]}$$

$$\frac{R(\mathtt{r}) = (1,-)}{\begin{array}{l}S \uplus \{\mathtt{l} \mapsto -\},R,T[(\mathtt{loc}\ \mathtt{l}\ \mathtt{at}\ \mathtt{r})\ \mathtt{:=}\ v] \\ \qquad \to S \uplus \{\mathtt{l} \mapsto v\},R,T[v]\end{array}}\ \text{[M3-e]} \qquad \frac{}{S,R,(\vec{e_1},E[\mathtt{fork}\ e_2],\vec{e_3}) \to S,R,(\vec{e_1},E[\mathtt{unit}],\vec{e_3},e_2)}\ \text{[S-e]}$$

Figure 10: Extended small region language: reduction rules

A source program e is well-typed iff $\{\mathtt{X_{sr}}\};\{\mathtt{sr} \mapsto \mathit{reg}(\mathtt{X_{sr}})@\mathtt{X_{sr}}\} \vdash e : \tau;\{\mathtt{X_{sr}}\}$. Figure 12 shows the additional rules for typing the intermediate expressions. The type environment is extended to map locations to types (e.g., $\Gamma(\mathtt{l}) = \tau$).

A.1 Proof of Safety

We reformulate the proof for the extended system. We also fill some of the details omitted from the proof in Section 2.4.

All expressions mentioned in the rest of the section are assumed to be in the intermediate language unless specified otherwise.

Definition 2 *A state $(S,R,\vec{e})$ is well-typed under the environment $\Delta;\Gamma$ iff*

(1) $\mathit{dom}(\Gamma) = \mathit{dom}(R) \cup \mathit{dom}(S)$, $\mathit{fvars}(\mathit{ran}(\Gamma)) \subseteq \Delta$, and $\Delta;\Gamma \vdash e_i : -;\{\mathtt{X_{sr}}\}$ for each $e_i \in \{\vec{e}\}$.

*(2) For each subexpression of the form $\Lambda\bullet\lambda - {:}\bullet.e_1$ **at** $-$ of $\vec{e}$, e_1 does not contain a subexpression of the form **inuse** $-$ **in** $-$.*

*(3) For each $\mathtt{r} \in \mathit{dom}(R)$ such that $\mathtt{r} \neq \mathtt{r_{sr}}$, the number of occurrences of subexpressions of the form **inuse** (**reg** $\mathtt{r}$ **at** $-$) **in** $-$ of $\vec{e}$ is equal to i where $R(\mathtt{r}) = (-,i)$.*

$$\frac{\Gamma(\mathtt{x}) = \tau}{\Delta;\Gamma \vdash \mathtt{x} : \tau; L}\ (\text{V-e}) \qquad \frac{\Delta;\Gamma \vdash e_1 : \tau_1; L \quad \Delta;\Gamma \uplus \{\mathtt{x} \mapsto \tau_1\} \vdash e_2 : \tau_2; L}{\Delta;\Gamma \vdash \mathtt{let\ x\ =\ } e_1 \ \mathtt{in}\ e_2 : \tau_2; L}\ (\text{L-e})$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_2 : \mathit{reg}(\mathtt{X})@\mathtt{Z}; L_2 \quad \mathit{fvars}(\tau_1) \cup \mathit{fvars}(L_1) \subseteq \Delta \uplus \{\vec{\alpha}\} \\ \Delta \uplus \{\vec{\alpha}\}; \Gamma \uplus \{x \mapsto \tau_1\} \vdash e_1 : \tau_2; L_1 \quad \mathtt{X}, \mathtt{Z} \in L_2\end{array}}{\Delta;\Gamma \vdash \Lambda\vec{\alpha}\lambda\mathtt{x}{:}\tau_1.e_1 \ \mathtt{at}\ e_2 : (\forall\vec{\alpha}.\tau_1 \xrightarrow{L_1} \tau_2)@\mathtt{X}; L_2}\ (\text{F1-e}) \qquad \frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : (\forall\vec{\alpha}.\tau_1 \xrightarrow{L_1} \tau_2)@\mathtt{X}; L_2 \quad \mathit{fvars}(\vec{\gamma}) \subseteq \Delta \\ \Delta;\Gamma \vdash e_2 : \tau_1[\vec{\gamma}/\vec{\alpha}]; L_2 \quad L_1[\vec{\gamma}/\vec{\alpha}] \cup \{\mathtt{X}\} \subseteq \mathtt{L}_2\end{array}}{\Delta;\Gamma \vdash e_1[\vec{\gamma}]\ e_2 : \tau_2[\vec{\gamma}/\vec{\alpha}]; L_2}\ (\text{F2-e})$$

$$\frac{\Delta;\Gamma \vdash \vec{e} : \vec{\tau}; L \quad \Delta;\Gamma \vdash e : \mathit{reg}(\mathtt{X})@\mathtt{Y}; L \quad \mathtt{X}, \mathtt{Y} \in L}{\Delta;\Gamma \vdash \vec{e}\ \mathtt{at}\ e : \vec{\tau}@\mathtt{X}; L}\ (\text{T1-e}) \qquad \frac{\Delta;\Gamma \vdash e : (\tau_1, \tau_2, ..., \tau_i, ...)@\mathtt{X}; L \quad \mathtt{X} \in L}{\Delta;\Gamma \vdash e.i : \tau_i; L}\ (\text{T2-e})$$

$$\frac{\Delta;\Gamma \vdash e : \tau[\gamma/\alpha]; L}{\Delta;\Gamma \vdash \mathtt{pack}\ e\ \mathtt{as}\ \exists\alpha.\tau : \exists\alpha.\tau; L}\ (\text{E1-e}) \qquad \frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \exists\alpha.\tau_1; L \quad \mathit{fvars}(\tau_1) \subseteq \Delta \uplus \{\alpha\} \\ \Delta \uplus \{\alpha\}; \Gamma \uplus \{\mathtt{x} \mapsto \tau_1\} \vdash e_2 : \tau_2; L \quad \mathit{fvars}(\tau_2) \subseteq \Delta\end{array}}{\Delta;\Gamma \vdash \mathtt{open\ x\ =\ } e_1\ \mathtt{as}\ \tau_1\ \mathtt{in}\ e_2 : \tau_2; L}\ (\text{E2-e})$$

$$\frac{\Delta;\Gamma \vdash e : \mathit{reg}(\mathtt{Y})@\mathtt{Z}; L \quad \mathtt{Y}, \mathtt{Z} \in L \quad \mathtt{X} \neq \mathtt{Y}}{\Delta;\Gamma \vdash \mathtt{newregion\ at}\ e : \exists\mathtt{X}.\mathit{reg}(\mathtt{X})@\mathtt{Y}; L}\ (\text{R1-e}) \qquad \frac{\Delta;\Gamma \vdash e : \mathit{reg}(\mathtt{X})@\mathtt{Y}; L \quad \mathtt{Y} \in L}{\Delta;\Gamma \vdash \mathtt{freeregion}\ e : \mathit{reg}(\mathtt{X})@\mathtt{Y}; L}\ (\text{R2-e})$$

$$\frac{\Delta;\Gamma \vdash e_1 : \mathit{reg}(\mathtt{X})@\mathtt{Y}; L \quad \Delta;\Gamma \vdash e_2 : \tau; L \cup \{\mathtt{X}\} \quad \mathtt{Y} \in L}{\Delta;\Gamma \vdash \mathtt{useregion}\ e_1\ \mathtt{in}\ e_2 : \tau; L}\ (\text{R3-e})$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau; L \quad \Delta;\Gamma \vdash e_2 : \mathit{reg}(\mathtt{X})@\mathtt{Y}; L \quad \mathtt{X}, \mathtt{Y} \in L}{\Delta;\Gamma \vdash \mathtt{ref}\ e_1\ \mathtt{at}\ e_2 : \mathit{ref}(\tau)@\mathtt{X}; L}\ (\text{M1-e}) \qquad \frac{\Delta;\Gamma \vdash e : \mathit{ref}(\tau)@\mathtt{X}; L \quad \mathtt{X} \in L}{\Delta;\Gamma \vdash \mathtt{!}e : \tau; L}\ (\text{M2-e})$$

$$\frac{\Delta;\Gamma \vdash e_1 : \mathit{ref}(\tau)@\mathtt{X}; L \quad \Delta;\Gamma \vdash e_2 : \tau; L \quad \mathtt{X} \in L}{\Delta;\Gamma \vdash e_1 \mathtt{\ :=\ } e_2 : \tau; L}\ (\text{M3-e}) \qquad \frac{\Delta;\Gamma \vdash e : \tau; \{\mathtt{X}_{\mathtt{sr}}\}}{\Delta;\Gamma \vdash \mathtt{fork}\ e : \mathit{unit}; L}\ (\text{S-e})$$

Figure 11: Extended small region language: type checking rules

(4) If $R(\mathtt{r}) = (b, i)$ and $i > 0$ then $b = 1$.

(5) For each $\mathtt{l} \in \mathit{dom}(S)$, $\Delta;\Gamma \vdash S(\mathtt{l}) : \Gamma(\mathtt{l}); -$.

We write $\Delta;\Gamma \vdash (S, R, \vec{e})$ to mean that $(S, R, \vec{e})$ is well typed under $\Delta;\Gamma$. Also for convenience, we write $\Delta \vdash \Gamma$ to mean $\mathit{fvars}(\mathit{ran}(\Gamma)) \subseteq \Delta$. Note that $\Delta;\Gamma \vdash (S, R, \vec{e})$ implies $\Delta \vdash \Gamma$.

Lemma 2 *If $\Delta;\Gamma \vdash e : \tau; -$ and $\Delta \vdash \Gamma$ then $\Delta \vdash \tau$.*

Proof: By induction on the type checking derivation. □

Lemma 3 (Substitution) *If $\Delta \uplus \{\vec{\alpha}\}; \Gamma \uplus \{\mathtt{x} \mapsto \tau_1\} \vdash e : \tau_2; L$, $\Delta \vdash \Gamma$ and $\Delta;\Gamma \vdash v : \tau_1[\vec{\gamma}/\vec{\alpha}]; -$, then $\Delta;\Gamma \vdash e[v/\mathtt{x}] : \tau_2[\vec{\gamma}/\vec{\alpha}]; L[\vec{\gamma}/\vec{\alpha}]$.*

Proof: By induction on the type checking derivation. □

Lemma 4 (Replacement) *Suppose $\Delta;\Gamma \vdash E[e_1] : -; L_1$. Let $\Delta;\Gamma \vdash e_1 : \tau; L_2$ be the subderivation. If $\Delta;\Gamma \vdash e_2 : \tau; L_2$, then $\Delta;\Gamma \vdash E[e_2] : -; L_1$.*

Proof: By induction on the type checking derivation. □

Lemma 5 *If $\Delta_1;\Gamma_1 \vdash e; \tau; L_1$, $\Delta_1 \subseteq \Delta_2$, $\Gamma_1 \subseteq \Gamma_2$, and $L_1 \subseteq L_2$, then $\Delta_2;\Gamma_2 \vdash e; \tau; L_2$.*

Proof: Trivial. □

Theorem 3 (Subject Reduction) *If $\Delta_1;\Gamma_1 \vdash (S_1, R_1, e_1)$ and $S_1, R_1, e_1 \rightarrow S_2, R_2, e_2$, then there exists Δ_2, Γ_2 such that $\Delta_2;\Gamma_2 \vdash (S_2, R_2, e_2)$.*

$$\frac{\begin{array}{c}\Delta;\Gamma\vdash e_2 : \mathit{reg}(\mathtt{X})@\mathtt{Z}; L_2 \qquad \mathit{fvars}(\tau_1)\cup\mathit{fvars}(L_1)\subseteq\Delta\uplus\{\vec{\alpha}\}\\ \Delta\uplus\{\vec{\alpha}\};\Gamma\uplus\{x\mapsto\tau_1\}\vdash e_1 : \tau_2; L_1 \qquad \mathtt{X},\mathtt{Z}\in L_2\end{array}}{\Delta;\Gamma\vdash \Lambda\bullet\lambda\mathtt{x}{:}\bullet.e_1\ \mathtt{at}\ e_2 : (\forall\vec{\alpha}.\tau_1\xrightarrow{L_1}\tau_2)@\mathtt{X}; L_2}\ \text{(F3-e)}\qquad \frac{\begin{array}{c}\Delta;\Gamma\vdash e_1 : (\forall\vec{\alpha}.\tau_1\xrightarrow{L_1}\tau_2)@\mathtt{X}; L_2 \qquad fvars(\vec{\gamma})\subseteq\Delta\\ \Delta;\Gamma\vdash e_2 : \tau_1[\vec{\gamma}/\vec{\alpha}]; L_2 \qquad L_1[\vec{\gamma}/\vec{\alpha}]\cup\{\mathtt{X}\}\subseteq \mathtt{L}_2\end{array}}{\Delta;\Gamma\vdash e_1[\bullet]\ e_2 : \tau_2[\vec{\gamma}/\vec{\alpha}]; L_2}\ \text{(F4-e)}$$

$$\frac{\Delta;\Gamma\vdash e : \tau[\gamma/\alpha]; L}{\Delta;\Gamma\vdash \mathtt{pack}\ e\ \mathtt{as}\ \bullet : \exists\alpha.\tau; L}\ \text{(E3-e)}\qquad \frac{\begin{array}{c}\Delta;\Gamma\vdash e_1 : \exists\alpha.\tau_1; L \qquad \mathit{fvars}(\tau_1)\subseteq\Delta\uplus\{\alpha\}\\ \Delta\uplus\{\alpha\};\Gamma\uplus\{\mathtt{x}\mapsto\tau_1\}\vdash e_2 : \tau_2; L \qquad \mathit{fvars}(\tau_2)\subseteq\Delta\end{array}}{\Delta;\Gamma\vdash \mathtt{open}\ \mathtt{x}\ \mathtt{=}\ e_1\ \mathtt{as}\ \bullet\ \mathtt{in}\ e_2 : \tau_2; L}\ \text{(E4-e)}$$

$$\frac{\Gamma(\mathtt{r})=\mathtt{X}\qquad \Gamma(\mathtt{s})=\mathtt{Y}\qquad \Delta;\Gamma\vdash e:\tau; L\cup\{\mathtt{X}\}\qquad \mathtt{Y}\in L}{\Delta;\Gamma\vdash \mathtt{inuse}\ (\mathtt{reg}\ \mathtt{r}\ \mathtt{at}\ \mathtt{s})\ \mathtt{in}\ e : \tau; L}\ \text{(R4-e)}\qquad \frac{}{\Delta;\Gamma\vdash \mathtt{abort} : \tau; L}\ \text{(A-e)}\qquad \frac{}{\Delta;\Gamma\vdash \mathtt{unit} : \mathit{unit}; L}\ \text{(U-e)}$$

$$\frac{\begin{array}{c}\mathit{fvars}(\tau_1)\cup\mathit{fvars}(L_1)\subseteq\Delta\uplus\{\vec{\alpha}\}\\ \Delta\uplus\{\vec{\alpha}\};\Gamma\uplus\{x\mapsto\tau_1\}\vdash e : \tau_2; L_1 \qquad \Gamma(\mathtt{r})=\mathtt{X}\end{array}}{\Delta;\Gamma\vdash \Lambda\bullet\lambda\mathtt{x}{:}\bullet.e\ \mathtt{at}\ \mathtt{r} : (\forall\vec{\alpha}.\tau_1\xrightarrow{L_1}\tau_2)@\mathtt{X}; L_2}\ \text{(FV-e)}\qquad \frac{\Delta;\Gamma\vdash\vec{v}:\vec{\tau}; L\qquad \Gamma(\mathtt{r})=\mathtt{X}}{\Delta;\Gamma\vdash \vec{v}\ \mathtt{at}\ \mathtt{r} : \vec{\tau}@\mathtt{X}; L}\ \text{(TV-e)}$$

$$\frac{\Gamma(\mathtt{r})=\mathtt{X}\qquad \Gamma(\mathtt{s})=\mathtt{Y}}{\Delta;\Gamma\vdash \mathtt{reg}\ \mathtt{r}\ \mathtt{at}\ \mathtt{s} : \mathit{reg}(\mathtt{X})@\mathtt{Y}; L}\ \text{(RV-e)}\qquad \frac{\Gamma(\mathtt{l})=\tau\qquad \Gamma(r)=\mathtt{X}}{\Delta;\Gamma\vdash \mathtt{loc}\ \mathtt{l}\ \mathtt{at}\ \mathtt{r} : \mathit{ref}(\tau)@\mathtt{X}; L}\ \text{(MV-e)}$$

Figure 12: Extended small region language: additional type rules for intermediate expressions

Proof: The conditions (2), (3), and (4) can be proved independent of the choice of Δ_2, Γ_2. The condition (2) follows from inspection of the reduction rules. Given (2), (3) is straightforward by inspection of [R4-e] and [R6-e]. The condition (4) is straightforward by inspection of [R4-e].

It remains to find Δ_2 and Γ_2 to satisfy (1) and (5). We do this via case analysis on the reduction rules.

[F1-e] Let $\Delta_2 = \Delta_1$ and $\Gamma_2 = \Gamma_1$. Let

$$E[\Lambda\bullet\lambda\mathtt{x}{:}\bullet.e\ \mathtt{at}\ (\mathtt{reg}\ \mathtt{r}\ \mathtt{at}\ \mathtt{s})]\in\{\vec{e_1}\}$$

be the active thread. Then by inspection of the type checking rules, there is a subderivation

$$\frac{\begin{array}{c}\Delta_2;\Gamma_2\vdash \mathtt{reg}\ \mathtt{r}\ \mathtt{at}\ \mathtt{s} : \mathit{reg}(\mathtt{X})@\mathtt{Z}; L_2\\ \mathit{fvars}(\tau_1)\cup\mathit{fvars}(L_1)\subseteq\Delta_2\uplus\{\vec{\alpha}\}\\ \Delta_2\uplus\{\vec{\alpha}\};\Gamma_2\uplus\{x\mapsto\tau_1\}\vdash e : \tau_2; L_1 \qquad \mathtt{X},\mathtt{Z}\in L_2\end{array}}{\Delta_2;\Gamma_2\vdash \Lambda\bullet\lambda\mathtt{x}{:}\bullet.e\ \mathtt{at}\ (\mathtt{reg}\ \mathtt{r}\ \mathtt{at}\ \mathtt{s}) : (\forall\vec{\alpha}.\tau_1\xrightarrow{L_1}\tau_2)@\mathtt{X}; L_2}$$

By inspection of (RV-e), it must be the case that $\Gamma_2(\mathtt{r}) = \mathtt{X}$. Hence by (FV-e),

$$\Delta_2;\Gamma_2\vdash \Lambda\bullet\lambda\mathtt{x}{:}\bullet.e\ \mathtt{at}\ \mathtt{r} : (\forall\vec{\alpha}.\tau_1\xrightarrow{L_1}\tau_2)@\mathtt{X}; L_2$$

By Lemma 4, it follows that

$$\Delta_2;\Gamma_2\vdash E[\Lambda\bullet\lambda\mathtt{x}{:}\bullet.e\ \mathtt{at}\ \mathtt{r}] : -; \{\mathtt{X}_{\mathtt{sr}}\}$$

Hence (1) holds. (5) is trivial.

[F2-e] Let $\Delta_2 = \Delta_1$ and $\Gamma_2 = \Gamma_2$. Let

$$E[(\Lambda\bullet\lambda\mathtt{x}{:}\bullet.e\ \mathtt{at}\ \mathtt{r})[\bullet]\ v]\in\{\vec{e_1}\}$$

be the active thread. Then by inspection of the type checking rules, there is a subderivation

$$\frac{\begin{array}{c}\Delta_2;\Gamma_2 \vdash (\Lambda\bullet\lambda\mathtt{x}{:}\bullet.e\ \mathtt{at}\ \mathtt{r}) : (\forall\vec{\alpha}.\tau_1 \xrightarrow{L_1} \tau_2)@\mathtt{X}; L_2 \\ \Delta_2;\Gamma_2 \vdash v : \tau_1[\vec{\gamma}/\vec{\alpha}]; L_2 \\ L_1[\vec{\gamma}/\vec{\alpha}] \cup \{\mathtt{X}\} \subseteq \mathtt{L_2} \qquad \mathit{fvars}(\vec{\gamma}) \subseteq \Delta_2\end{array}}{\Delta_2;\Gamma_2 \vdash (\Lambda\bullet\lambda\mathtt{x}{:}\bullet.e\ \mathtt{at}\ \mathtt{r})[\bullet]\ v : \tau_2[\vec{\gamma}/\vec{\alpha}]; L_2}$$

By inspection of (FV-e), it must be the case that

$$\Delta_2 \uplus \{\vec{\alpha}\};\Gamma_2 \uplus \{x \mapsto \tau_1\} \vdash e : \tau_2; L_1$$

Hence by Lemma 3, it follows that

$$\Delta_2;\Gamma_2 \vdash e[v/x] : \tau_2[\vec{\gamma}/\vec{\alpha}]; L_1[\vec{\gamma}/\vec{\alpha}]$$

By Lemma 5,

$$\Delta_2;\Gamma_2 \vdash e[v/x] : \tau_2[\vec{\gamma}/\vec{\alpha}]; L_2$$

By Lemma 4, it follows that

$$\Delta_2;\Gamma_2 \vdash E[e[v/x]] : -; \{\mathtt{X_{sr}}\}$$

Hence (1) holds. (5) is trivial.

[E-e] Let $\Delta_2 = \Delta_1$ and $\Gamma_2 = \Gamma_1$. Let

$$E[\mathtt{open\ x\ =\ (pack}\ v\ \mathtt{as}\ \bullet\mathtt{)\ as}\ \bullet\ \mathtt{in}\ e] \in \{\vec{e_1}\}$$

be the active thread. Then by inspection of the type checking rules, there is a subderivation

$$\frac{\begin{array}{c}\Delta_2;\Gamma_2 \vdash \mathtt{pack}\ v\ \mathtt{as}\ \bullet : \exists\alpha.\tau_1; L \qquad \mathit{fvars}(\tau_1) \subseteq \Delta_2 \uplus \{\alpha\} \\ \Delta_2 \uplus \{\alpha\};\Gamma_2 \uplus \{\mathtt{x} \mapsto \tau_1\} \vdash e : \tau_2; L \qquad \mathit{fvars}(\tau_2) \subseteq \Delta_2\end{array}}{\Delta_2;\Gamma_2 \vdash \mathtt{open\ x\ =\ (pack}\ v\ \mathtt{as}\ \bullet\mathtt{)\ as}\ \bullet\ \mathtt{in}\ e : \tau_2; L}$$

By inspection of (E3-e), it must be the case that $\Delta_2;\Gamma_2 \vdash v : \tau[\gamma/\alpha]; L$. Hence by Lemma 3, it follows that

$$\Delta_2;\Gamma_2 \vdash e[v/x] : \tau_2[\vec{\gamma}/\vec{\alpha}]; L[\vec{\gamma}/\vec{\alpha}]$$

We have $\tau_2[\vec{\gamma}/\vec{\alpha}] = \tau_2$. Since the subderivation appears in the derivation having $\{\mathtt{X_{sr}}\}$ as its effect set, it is easy to see that $\mathit{fvars}(L) \subseteq \Delta_2$. So $L[\vec{\gamma}/\vec{\alpha}] = L$. Hence we have

$$\Delta_2;\Gamma_2 \vdash e[v/x] : \tau_2; L$$

By Lemma 4, it follows that

$$\Delta_2;\Gamma_2 \vdash E[e[v/x]] : -; \{\mathtt{X_{sr}}\}$$

Hence (1) holds. (5) is trivial.

[R1-e] Let $\mathtt{W} \notin \Delta_1$. Let $\Delta_2 = \Delta_1 \uplus \{\mathtt{W}\}$ and $\Gamma_2 = \Gamma_1 \uplus \{\mathtt{t} \mapsto \mathtt{W}\}$. It is easy to see that $\mathit{dom}(\Gamma_2) = \mathit{dom}(R \uplus \{\mathtt{t} \mapsto (1,0)\}) \cup \mathit{dom}(S)$ and $\mathit{fvars}(\mathit{ran}(\Gamma_2)) \subseteq \Delta_2$.

For each thread $e \in \{\vec{e_1}\}$, $\Delta_2;\Gamma_2 \vdash e : -; \{\mathtt{X_{sr}}\}$ by Lemma 5.

Let

$$E[\mathtt{newregion\ at\ (reg\ r\ at\ s)}] \in \{\vec{e_1}\}$$

be the active thread. Then by inspection of the type checking rules, there is a subderivation

$$\frac{\Delta_2;\Gamma_2 \vdash \mathtt{reg\ r\ at\ s} : \mathit{reg}(\mathtt{Y})@\mathtt{Z}; L \qquad \mathtt{Y},\mathtt{Z} \in L \qquad \mathtt{X} \neq \mathtt{Y}}{\Delta_2;\Gamma_2 \vdash \mathtt{newregion\ at\ (reg\ r\ at\ s)} : \exists\mathtt{X}.\mathit{reg}(\mathtt{X})@\mathtt{Y}; L}$$

By inspection of (RV-e), it must be the case that $\Gamma_2(\texttt{r}) = \texttt{Y}$. Hence by (RV-e) again $\Delta_2, \Gamma_2 \vdash \texttt{reg t at r} : \mathit{reg}(\texttt{W})@\texttt{Y}$. By (E3-e),

$$\Delta_2; \Gamma_2 \vdash \texttt{pack (reg t at r) as} \bullet : \exists \texttt{X}.\mathit{reg}(\texttt{X})@\texttt{Y}; L$$

By Lemma 4, it follows that

$$\Delta_2; \Gamma_2 \vdash E[\texttt{pack (reg t at r) as} \bullet] : -; \{\texttt{X}_{\texttt{sr}}\}$$

Hence (1) holds. (5) is trivial.

[R2-e] Let $\Delta_2 = \Delta_1$ and $\Gamma_2 = \Gamma_1$. Let

$$E[\texttt{freeregion (reg r at s)}] \in \{\vec{e_1}\}$$

be the active thread. Then by inspection of the type checking rules, there is a subderivation

$$\frac{\Delta_2; \Gamma_2 \vdash \texttt{reg r at s} : \mathit{reg}(\texttt{X})@\texttt{Y}; L \qquad \texttt{Y} \in L}{\Delta_2; \Gamma_2 \vdash \texttt{freeregion (reg r at s)} : \mathit{reg}(\texttt{X})@\texttt{Y}; L}$$

Hence $\Delta_2; \Gamma_2 \vdash \texttt{reg r at s} : \mathit{reg}(\texttt{X})@\texttt{Y}; L$. By Lemma 4, it follows that

$$\Delta_2; \Gamma_2 \vdash E[\texttt{reg r at s}] : -; \{\texttt{X}_{\texttt{sr}}\}$$

Hence (1) holds. (5) is trivial.

[R3-e] Let $\Delta_2 = \Delta_1$ and $\Gamma_2 = \Gamma_1$. By (A-e),

$$\Delta_2; \Gamma_2 \vdash \texttt{abort} : -; \{\texttt{X}_{\texttt{sr}}\}$$

Hence (1) holds. (5) is trivial.

[R4-e] Let $\Delta_2 = \Delta_1$ and $\Gamma_2 = \Gamma_1$. Let

$$E[\texttt{useregion (reg r at s) in } e] \in \{\vec{e_1}\}$$

be the active thread. Then by inspection of the type checking rules, there is a subderivation

$$\frac{\begin{array}{c}\Delta_2; \Gamma_2 \vdash \texttt{reg r at s} : \mathit{reg}(\texttt{X})@\texttt{Y}; L \\ \Delta_2; \Gamma_2 \vdash e : \tau; L \cup \{\texttt{X}\} \qquad \texttt{Y} \in L\end{array}}{\Delta_2; \Gamma_2 \vdash \texttt{useregion (reg r at s) in } e : \tau; L}$$

By inspection of (RV-e), it must be the case that $\Gamma_2(\texttt{r}) = \texttt{X}$ and $\Gamma_2(\texttt{s}) = \texttt{Y}$. Hence by (R4-e),

$$\Delta_2; \Gamma_2 \vdash \texttt{inuse (reg r at s) in } e : \tau; L$$

By Lemma 4, it follows that

$$\Delta_2; \Gamma_2 \vdash E[\texttt{inuse (reg r at s) in } e] : -; \{\texttt{X}_{\texttt{sr}}\}$$

Hence (1) holds. (5) is trivial.

[R5-e] Similar to the case [R3-e].

[R6-e] Let $\Delta_2 = \Delta_1$ and $\Gamma_2 = \Gamma_1$. Let

$$E[\texttt{inuse (reg r at s) in } v] \in \{\vec{e_1}\}$$

be the active thread. Then by inspection of the type checking rules, there is a subderivation

$$\frac{\Gamma_2(\texttt{r}) = \texttt{X} \quad \Gamma_2(\texttt{s}) = \texttt{Y} \quad \Delta_2; \Gamma_2 \vdash v : \tau; L \cup \{\texttt{X}\} \quad \texttt{Y} \in L}{\Delta_2; \Gamma_2 \vdash \texttt{inuse (reg r at s) in } v : \tau; L}$$

Since v is a value, by inspection of the type checking rules, $\Delta_2; \Gamma_2 \vdash v : \tau; L$. By Lemma 4, it follows that

$$\Delta_2; \Gamma_2 \vdash E[v] : -; \{\texttt{X}_{\texttt{sr}}\}$$

Hence (1) holds. (5) is trivial.

[T1-e] Similar to the case [F1-e].

[T2-e] Similar to the case [F2-e].

[L-e] Similar to the case [F2-e].

[M1-e] Let $E[\texttt{ref } v \texttt{ at reg r at s}] \in \{\vec{e_1}\}$ be the active thread. Then by inspection of the type checking rules, there is a subderivation

$$\frac{\dfrac{\Delta_1; \Gamma_1 \vdash v : \tau; L}{\Delta_1; \Gamma_1 \vdash \texttt{reg r at s} : \mathit{reg}(\texttt{X})@\texttt{Y}; L} \quad \texttt{X}, \texttt{Y} \in L}{\Delta_1; \Gamma_1 \vdash \texttt{ref } v \texttt{ at reg r at s} : \mathit{ref}(\tau)@\texttt{X}; L}$$

Let $\Delta_2 = \Delta_1$ and $\Gamma_2 = \Gamma_1 \uplus \{\texttt{l} \mapsto \tau\}$. By Lemma 2, $\mathit{dom}(\Gamma_2) = \mathit{dom}(R) \cup \mathit{dom}(S \uplus \{\texttt{l} \mapsto v\})$ and $\mathit{fvars}(\mathit{ran}(\Gamma_2)) \subseteq \Delta_2$.

For each thread $e \in \{\vec{e_1}\}$, $\Delta_2; \Gamma_2 \vdash e : -; \{\texttt{X}_{\texttt{sr}}\}$ by Lemma 5. By inspection of (RV-e), it must be the case that $\Gamma_2(\texttt{r}) = \texttt{X}$. Hence by (MV-e),

$$\Delta_2; \Gamma_2 \vdash \texttt{loc l at r} : \mathit{ref}(\tau)@\texttt{X}; L$$

By Lemma 4, it follows that

$$\Delta_2; \Gamma_2 \vdash E[\texttt{loc l at r}] : -; \{\texttt{X}_{\texttt{sr}}\}$$

Hence (1) holds. Since $\Delta_2; \Gamma_2 \vdash v : \tau; L$, (5) holds.

[M2-e] Let $\Delta_2 = \Delta_1$ and $\Gamma_2 = \Gamma_2$. Let $E[\texttt{!(loc l at r)}] \in \{\vec{e_1}\}$ be the active thread. Then by inspection of the type checking rules, there is a subderivation

$$\frac{\Delta_2; \Gamma_2 \vdash \texttt{loc l at r} : \mathit{ref}(\tau)@\texttt{X}; L \quad \texttt{X} \in L}{\Delta_2; \Gamma_2 \vdash \texttt{!(loc l at r)} : \tau; L}$$

By inspection of (MV-e), it must be the case that $\Gamma_2(\texttt{l}) = \tau$. Hence by (5) (on the left state), $\Delta_2; \Gamma_2 \vdash v : \tau; L$. By Lemma 4, it follows that

$$\Delta_2; \Gamma_2 \vdash E[v] : -; \{\texttt{X}_{\texttt{sr}}\}$$

Hence (1) holds. (5) is trivial.

[M3-e] Let $\Delta_2 = \Delta_1$ and $\Gamma_2 = \Gamma_2$. Let

$$E[(\texttt{loc l at r}) \texttt{ := } v] \in \{\vec{e_1}\}$$

be the active thread. Then by inspection of the type checking rules, there is a subderivation

$$\frac{\begin{array}{c}\Delta_2; \Gamma_2 \vdash \texttt{loc l at r} : \mathit{ref}(\tau)@\texttt{X}; L \\ \Delta_2; \Gamma_2 \vdash v : \tau; L \qquad \texttt{X} \in L\end{array}}{\Delta_2; \Gamma_2 \vdash (\texttt{loc l at r}) \texttt{ := } v : \tau; L}$$

By Lemma 4, it follows that

$$\Delta_2; \Gamma_2 \vdash E[v] : -; \{\texttt{X}_{\texttt{sr}}\}$$

Hence (1) holds.

By inspection of (MV-e), it must be the case that $\Gamma_2(\texttt{l}) = \tau$. Hence (5) holds.

[S-e] Let $\Delta_2 = \Delta_1$ and $\Gamma_2 = \Gamma_2$. There is a subderivation for the thread $E[\texttt{fork}\, e_2]$

$$\frac{\Delta_2; \Gamma_2 \vdash e_2 : \tau; \{\texttt{X}_{\texttt{sr}}\}}{\Delta_2; \Gamma_2 \vdash \texttt{fork}\, e_2 : \mathit{unit}; L}$$

By (U-e), $\Delta_2; \Gamma_2 \vdash \texttt{unit} : \mathit{unit}; L$. By Lemma 4, it follows that

$$\Delta_2; \Gamma_2 \vdash E[\texttt{unit}] : -; \{\texttt{X}_{\texttt{sr}}\}$$

We also have $\Delta_2; \Gamma_2 \vdash e_2 : -; \{\texttt{X}_{\texttt{sr}}\}$. Hence (1) holds. (5) is trivial. □

Lemma 6 *If $\Delta; \Gamma \vdash E[e] : -; \{\texttt{X}_{\texttt{sr}}\}$ then there is a sub-derivation $\Delta; \Gamma \vdash e : -; L$ such that if $\texttt{X} \in L$ and $\texttt{X} \neq \texttt{X}_{\texttt{sr}}$ then there is an occurrence of* $\texttt{inuse (reg r at } -) \texttt{ in } -$ *in $E[e]$ such that $\Gamma(\texttt{r}) = \texttt{X}$.*

Proof: By inspection on the definition of evaluation contexts and the corresponding type checking rules. □

Theorem 4 (Progress) *If $\Delta; \Gamma \vdash (S, R, \vec{e})$ and not all of $\vec{e}$ is a value, then there is a reduction from $(S, R, \vec{e})$.*

Proof: The main part of the proof is in showing that condition of the form $R(\ldots) = \ldots$ in the hypotheses of each reduction rule is satisfied. These hypotheses appears across multiple reduction rules, but essentially the same argument can be used for all of them. We shall show this for the case $\vec{e}$ contains $E[\Lambda\bullet\lambda\texttt{x} : \bullet.e \texttt{ at (reg r at s)}]$.

By inspection of the type checking rules, there is a subderivation

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \texttt{reg r at s} : \mathit{reg}(\texttt{X})@\texttt{Z}; L_2 \\ \mathit{fvars}(\tau_1) \cup \mathit{fvars}(L_1) \subseteq \Delta \uplus \{\vec{\alpha}\} \\ \Delta \uplus \{\vec{\alpha}\}; \Gamma \uplus \{x \mapsto \tau_1\} \vdash e : \tau_2; L_1 \qquad \texttt{X}, \texttt{Z} \in L_2\end{array}}{\Delta; \Gamma \vdash \Lambda\bullet\lambda\texttt{x}{:}\bullet.e \texttt{ at (reg r at s)} : (\forall\vec{\alpha}.\tau_1 \xrightarrow{L_1} \tau_2)@\texttt{X}; L_2}$$

By Lemma 6, there is an occurrence of $\texttt{inuse (reg r at } -) \texttt{ in } -$ such that $\Gamma(\texttt{r}) = \texttt{X}$ and an occurrence of $\texttt{inuse (reg s at } -) \texttt{ in } -$ such that $\Gamma(\texttt{s}) = \texttt{Z}$. By Definition 2 (3) and (4), this implies that $R(\texttt{r}) = (1, -)$ and $R(\texttt{s}) = (1, -)$. Hence [F1-e] can be applied to reduce the state.

Other cases can be proved analogously. □

Theorem 5 *Let e_1 be a well-typed program in the source language. Let e_2 be its corresponding type-erased program in the intermediate language. Then $\{\texttt{X}_{\texttt{sr}}\}; \{\texttt{r}_{sr} \mapsto \texttt{X}_{\texttt{sr}}\}(\emptyset, \{\texttt{r}_{sr} \mapsto (1,1)\}, \texttt{e}_2)$ where $\texttt{r}_{sr}$ is the starting region.*

Proof: Trivial. □

Corollary 1 (Type Soundness) *If a source program e is well typed then it does not get stuck, that is, e is memory safe.*

Proof: By Theorem 3, Theorem 4 and Theorem 5 □