# Polymorphic Fractional Capabilities

Hirotoshi Yasuoka and Tachio Terauchi

Tohoku University
{yasuoka,terauchi}@kb.ecei.tohoku.ac.jp

**Abstract.** The capability calculus is a framework for statically reasoning about program resources such as deallocatable memory regions. Fractional capabilities, originally proposed by Boyland for checking the determinism of parallel reads in multi-thread programs, extend the capability calculus by extending the capabilities to range over the rational numbers. Fractional capabilities have since found numerous applications, including race detection, buffer bound inference, security analyses, and separation logic. However, previous work on fractional capability systems either lacked polymorphism or lacked an efficient inference procedure. Automated inference is important for the application of the calculus to static analysis. This paper addresses the issue by presenting a polymorphic fractional capability calculus that allows polynomial-time inference via a reduction to rational linear programming.

## 1 Introduction

The capability calculus [5] was originally proposed as a framework for region-based memory management, that is, a system for guaranteeing that a deallocated region is never accessed. The capability calculus is a good framework for statically reasoning about properties of general program resources such as memory regions [5], reference cells [4], and communication channels [14]. Researchers [4, 13] extended the framework to *fractional capabilities* ([4] called them *fractional permissions*), allowing **(1)** more flexibility by letting the capabilities range over rational numbers, which allows reasoning about concurrent reads and writes, and **(2)** efficient inference via a reduction to (rational) linear programming, whereas no efficient inference is known for the non-fractional calculi[1]. Fractional capabilities have found applications in many areas of program verification and program logic, including determinism checking [4, 12, 13], separation logic [3], security protocol analysis [2, 7], buffer bound inference [14], race detection [10], and secure information flow [11].

Previous work on capability calculus either lacked parametric polymorphism [14, 10, 11] or lacked an efficient inference procedure [5, 4, 3]. By parametric polymorphism, we mean allowing capabilities appearing in the function types to be parametrized by *resource variables*[2]. The issue turns out to be surprisingly non-trivial as "obvious" approaches result in an overly conservative system.

[1] Indeed, some of them are proven to be NP-hard [13, 7].

[2] Resource variables range over program resources to be reasoned by the calculus such as memory locations or communication channels, depending on the application.

The rest of the paper is organized as follows. In the next section, we give an overview of the main issues, highlighting the problems with naive approaches, and presenting our solution informally. Section 3 formally presents the solution and proves its soundness. Section 4 shows the type inference algorithm. The inference algorithm utilizes linear programming and runs in time polynomial in the size of the underlying Hindley-Milner types. To present the idea assuming little prior background in fractional capabilities, we focus on the application of the capability calculus to the problem of region-based memory management [5]. But, our idea can also be applied to other fractional capability calculi. Section 5 discusses how the idea can be transferred to the capability calculus for race detection [10]. Section 6 discusses related work and concludes. The appendix contains the proofs and the figures omitted from the main body of the paper.

## 2 Informal Overview

We informally present the idea by showing how the capability calculus guarantees region-based memory safety. A program is said to be *memory safe* if it does not access a deleted region. Consider the following program fragment.

$$
\begin{array}{l}
\texttt{let}\ a = \texttt{ref}\ 0@\rho_a\ \texttt{in} \\
\texttt{let}\ b = \texttt{ref}\ 0@\rho_b\ \texttt{in} \\
\texttt{let}\ f = \texttt{fun}\ f[\rho]\ (x : \mathit{ref}(\mathit{int})@\rho)\ = !x; \texttt{free}\ \rho\ \texttt{in} \\
\quad f[\rho_a]\ (a); f[\rho_b]\ (b)
\end{array}
$$

The program allocates two reference cells $a$ and $b$, hopefully in separate regions $\rho_a$ and $\rho_b$. The function $f$ is called with each cell, dereferencing the given cell and deleting its region. Here, $\rho$ is a region parameter that is passed from the caller. Unlike in the traditional stack-discipline of the Tofte-Talpin region calculus [15], the capability calculus allows non-scoped region deletion, which introduces more opportunities for dangling pointers. Here, ; is the sequential composition, $!e$ reads the reference cell $e$, and $\texttt{free}\ \rho$ deletes the region $\rho$.

Note that we need $f$ to be polymorphic in $\rho$ because otherwise, the regions $\rho_a$ and $\rho_b$ would be equated, and so the program becomes memory unsafe (as the second call to $f$ would try to read from the deleted region). Therefore, we would like to assign $f$ the following polymorphic type.

$$\forall\rho.(\{\rho \mapsto 1\}, \mathit{ref}(\mathit{int})@\rho) \to (\{\rho \mapsto 0\}, \mathit{int})$$

The type says that the function takes a region $\rho$ and an argument of the type $\mathit{ref}(\mathit{int})@\rho$ (i.e., an integer reference allocated in the region $\rho$) and returns an integer. It also says that the caller is required to have the *capability* $\{\rho \mapsto 1\}$. (Any capability greater than 0 is sufficient for dereferencing or writing to the region whereas the capability greater than or equal to 1 is needed to delete the region.) The function returns to the caller the capability $\{\rho \mapsto 0\}$ indicating that $\rho$ is deleted and cannot be accessed or deleted after the call.

```
letreg ρa in
  let a = ref 0@ρa in
  let f = fun f[ρx, ρy] (x : ref(int)@ρx, y : ref(int)@ρy) = !x; free ρx; !y
  in
  f[ρa, ρa] (a, a)
```

**Fig. 1.** A memory unsafe program.

Instantiating the polymorphic type at the calls, we get the types

$$(\{\rho_a \mapsto 1\}, ref(int)@\rho_a) \to (\{\rho_a \mapsto 0\}, int)$$
$$(\{\rho_b \mapsto 1\}, ref(int)@\rho_b) \to (\{\rho_b \mapsto 0\}, int)$$

and we are able to type check the program as memory safe, assuming that the capability before the code fragment is $\{\rho_a \mapsto 1, \rho_b \mapsto 1\}$, indicating that the code may access and delete the regions $\rho_a$ and $\rho_b$. The capability for a region $\rho$ is initialized to 1 when the region is created at `letreg` $\rho$.

Formally, a capability is a mapping from region variables ($\rho$'s) to non-negative rationals $[0, \infty)$ (e.g., $\{\rho_a \mapsto 1\}$ is a shorthand for a function that maps $\rho_a$ to 1 and all other region variables to 0). Note that instantiating the type of $f$ by $\rho_a$ and $\rho_b$ involve substitutions $\{\rho \mapsto 1\}[\rho_a/\rho] = \{\rho_a \mapsto 1\}$ and $\{\rho \mapsto 1\}[\rho_b/\rho] = \{\rho_b \mapsto 1\}$. Given a capability $\Psi$, the *naive instantiation* $\Psi[\rho_1/\rho_2]$ syntactically replaces $\rho_2$ in the capability $\Psi$ by $\rho_1$.

### 2.1 Additive Instantiation

Unfortunately, the naive instantiation is inadequate. Consider the program shown in Figure 1. Now, $f$ takes two cells, $x$ and $y$, accesses $x$, deletes the region in which $x$ is allocated (i.e., $\rho_x$), and then accesses $y$. Note that the program is unsafe because $f$ is called with both arguments set to $a$, and therefore, deletes the region where $a$ is allocated (i.e., $\rho_a$) before accessing $a$ for the second time.

The function $f$ can be given the type

$$\forall\rho_x.\forall\rho_y.(\Psi_{in}, ref(int)@\rho_x, ref(int)@\rho_y) \to (\Psi_{out}, int)$$

where $\Psi_{in} = \{\rho_x \mapsto 1, \rho_y \mapsto q\}$ and $\Psi_{out} = \{\rho_x \mapsto 0, \rho_y \mapsto q\}$ for some $q > 0$. When we instantiate the type via the substitution $\theta = [\rho_a/\rho_x][\rho_a/\rho_y]$ naively, the capability required before the call is $\Psi_{in}\theta = \{\rho_a \mapsto 1, \rho_a \mapsto q\}$ and the capability required after the call is $\Psi_{out}\theta = \{\rho_a \mapsto 0, \rho_a \mapsto q\}$. The capabilities are not mappings. To overcome the issue, the existing polymorphic fractional capability calculi (e.g. [4]) performs *additive instantiation*, defined as follows.

$$\Psi[\rho_1/\rho_2]_\oplus = \{\rho_1 \mapsto \Psi(\rho_2) + \Psi(\rho_1)\} \cup \{\rho_2 \mapsto 0\} \cup \{\rho \mapsto \Psi(\rho) \mid \rho \notin \{\rho_1, \rho_2\}\}$$

Then,

$$\Psi_{in}[\rho_a/\rho_x]_\oplus[\rho_a/\rho_y]_\oplus = \{\rho_a \mapsto 1, \rho_y \mapsto q\}[\rho_a/\rho_y]_\oplus = \{\rho_a \mapsto 1 + q\}$$

With this instantiation scheme, we are able to safely reject the program in Figure 1, because now the call $f[\rho_a, \rho_a]\,(a, a)$ requires $\{\rho_a \mapsto 1 + q\}$ where $q > 0$, but the caller only has $\{\rho_a \mapsto 1\}$.

### 2.2 Down Instantiation

The additive instantiation scheme discussed above, while sound, is somewhat conservative in the presence of recursive calls. Consider the following program.

$$
\begin{array}{l}
\texttt{letreg}\ \rho_a\ \texttt{in} \\
\quad \texttt{let}\ a = \texttt{ref}\ 0@\rho_a\ \texttt{in} \\
\quad \texttt{let}\ f = \\
\qquad \texttt{fun}\ f[\rho_x, \rho_y]\ (x : \mathit{ref}(\mathit{int})@\rho_x, y : \mathit{ref}(\mathit{int})@\rho_y) = !x; !y; f[\rho_x, \rho_x]\ (x, x) \\
\quad \texttt{in}\ \ f[\rho_a, \rho_a]\ (a, a)
\end{array}
$$

This program is obviously safe (because no region is deleted). However it is not typable with the additive instantiation scheme. To see this, note that $f$ must have a type of form $\forall \rho_x, \rho_y.(\Psi_{in}, \mathit{ref}(\mathit{int})@\rho_x, \mathit{ref}(\mathit{int})@\rho_y) \rightarrow (\Psi_{out}, \mathit{int})$ with $\Psi_{in}(\rho_x) > 0$ and $\Psi_{in}(\rho_y) > 0$ because $f$ accesses both $x$ and $y$. But, the capability

$$\Psi_{rec} = \Psi_{in}[\rho_x/\rho_x]_{\oplus}[\rho_y/\rho_x]_{\oplus} = \Psi_{in}[\rho_y/\rho_x]_{\oplus}$$

is required before the recursive call $f[\rho_x, \rho_x]\,(x, x)$. Thus, $\Psi_{rec}(\rho_x) = \Psi_{in}(\rho_x) + \Psi_{in}(\rho_y)$. But because $\Psi_{in} = \Psi_{rec}$ (or $\Psi_{in} \geq \Psi_{rec}$[3]), we have

$$\Psi_{in}(\rho_x) \geq \Psi_{in}(\rho_x) + \Psi_{in}(\rho_y) \quad \Psi_{in}(\rho_x) > 0 \quad \Psi_{in}(\rho_y) > 0$$

It is easy to see that there exists no non-negative rational number that can be assigned to $\Psi_{in}(\rho_x)$ to satisfy these inequalities. Therefore, the type system rejects the program as untypable.

One way to overcome the issue is to allow polymorphism over the fractions as done in some fractional capability calculi [4, 3]. Unfortunately, it is unclear whether an efficient inference exists for such systems.

Instead, we propose to relax the caller's requirement when the function does not delete the region. Specifically, when a function requests a positive capability for some $\rho$, we allow a call to the function to be type checked with a positive capability for $\rho$ that is lower than the actual capability requested by the callee, provided that the call does not delete $\rho$. The rationale for this is that any positive capability is sufficient for a region access (but not region deletion). To this end, we introduce a new instantiation scheme called *down instantiation*, defined as follows.

$$\Psi[\rho_1/\rho_2]_{\Downarrow} = \{\rho_1 \mapsto \frac{1}{2}(\Psi(\rho_2) + \Psi(\rho_1))\} \cup \{\rho \mapsto \Psi(\rho) \mid \rho \notin \{\rho_1, \rho_2\}\}$$

[3] This constraint is from the fact that a sequential composition passes capabilities along the sequence.

$$
\begin{array}{l}
\rho \in \mathbf{Regvar} \\
\Psi \in \mathbf{Regvar} \to [0, \infty) \\
L ::= \emptyset \mid L \cup \{\rho\} \\
\tau ::= \mathit{int} \mid \mathit{ref}(\tau)@\rho \mid (\Psi_{in}, \tau_{in}) \xrightarrow{L} (\Psi_{out}, \tau_{out}) \mid \forall \rho.\tau \\
\\
v ::= \lambda x : \tau.e \mid \Lambda \rho.v \\
e ::= n \mid x \mid v \mid \mathtt{fix}\, x : \tau.v \mid e\,(e') \mid e[\rho] \\
\quad \mid \mathtt{ref}\, e@\rho \mid !e \mid e := e' \mid \mathtt{letreg}\, \rho\, \mathtt{in}\, e \mid \mathtt{free}\, \rho
\end{array}
$$

**Fig. 2.** The syntax of the region language.

Note that the down instantiation *lowers* the capability by one half for the region being instantiated.

Using the down instantiation in the running example, linear inequalities to be solved becomes as follows.

$$\Psi_{in}(\rho_x) \geq \frac{1}{2}(\Psi_{in}(\rho_x) + \Psi_{in}(\rho_y)) \quad \Psi_{in}(\rho_x) > 0 \quad \Psi_{in}(\rho_y) > 0$$

These inequalities are solvable (e.g., by assigning 1 to both $\Psi_{in}(\rho_x)$ and $\Psi_{in}(\rho_y)$), and hence, the program becomes typable.

For soundness, we may only apply the down instantiation to regions that are not deleted by the call because it would be unsafe to lower a capability required for a region deletion to some positive number less than 1. Therefore, we select the instantiation scheme based on which region a function call may delete. We use the additive instantiation when instantiating regions that are deleted, and use the down instantiation for those that are not. We use *effects* to infer the deleted regions. In the example above, the effect analysis detects that $f$ frees neither $\rho_x$ nor $\rho_y$, and therefore, that the down instantiation can be used for $\rho_x$ and $\rho_y$.

## 3 Polymorphic Fractional Capability Calculus

We now formally define the polymorphic fractional capability calculus. We focus on the simple region language shown in Figure 2. The language is essentially the standard Tofte-Talpin region language [15] extended with $\mathtt{free}\,\rho$ and function types containing capabilities. Note that $\mathtt{free}\rho$ can be used to free a region before its scope expires (i.e., before $e$ is fully evaluated in $\mathtt{letreg}\,\rho\,\mathtt{in}\,e$), thus possibly creating a dangling pointer.

We briefly describe the syntax of expressions ($e$). Expressions include integer constants $n$, variables $x$, functions $\lambda x : \tau.e$, region variable abstractions $\Lambda\rho.v$, recursive definitions $\mathtt{fix}\, x : \tau.v$, function applications $e\,(e')$, region applications $e[\rho]$, reference allocations $\mathtt{ref}\, e@\rho$, reference reads $!e$, reference writes $e := e'$, region allocations $\mathtt{letreg}\, \rho\, \mathtt{in}\, e$, and region deallocations $\mathtt{free}\, \rho$. Region abstraction and recursive definition are restricted to *values* $v$, which, for now, are just functions (and their region abstractions). For simplicity, the region language

only allocates reference cells in regions, but it is easy to extend the language so that function closures are also allocated in regions. We write $\texttt{let}\ x : \tau = e_1\ \texttt{in}\ e_2$ for $(\lambda x : \tau.e_2)\,(e_1)$, and $e_1; e_2$ for $\texttt{let}\ x : \tau = e_1\ \texttt{in}\ e_2$ where $x$ is not free in $e_2$.

Each function takes a single argument, but a multi-argument function can be encoded via currying. Instead of letting functions directly take region arguments as in Section 2, we use separate syntax for region variable abstraction and region instantiation. We also use separate syntax for recursive definition. For example, $\texttt{fun}\ f[\rho]\ (x : \tau) = e$ from Section 2 can be expressed as $\texttt{fix}\ f : \tau'.\Lambda\rho.\lambda x : \tau.e$. Multi-region arguments can also be encoded by currying.

Next, we describe the grammar of the types. The types include integer types *int*, reference types $\mathit{ref}(\tau)@\rho$, region polymorphic types $\forall\rho.\tau$, and function types $(\Psi_{in}, \tau_{in}) \xrightarrow{L} (\Psi_{out}, \tau_{out})$ where $L$ denotes the latent effect. Unlike in the Tofte-Talpin region system, we do not use effects to control the access to regions. As explained in Section 2, the latent effect overapproximates the regions that may be deleted by calling the function.

$\Psi$ denotes a *capability*, which as discussed above, is a function from region variables **Regvar** to non-negative rational numbers in the range $[0, \infty)$. As discussed in Section 2, a capability represents the access rights over the regions. Informally, having $\Psi$ such that $\Psi(\rho) > 0$ means that the region $\rho$ can be accessed (i.e., allocated, read, or written). In addition, $\Psi(\rho) \geq 1$ means that the region $\rho$ can also be deleted. Deleting a region *consumes* the capability, that is, the capability for $\rho$ after $\texttt{free}\,\rho$ becomes 0. The capability for a region is initialized to 1 when the region is created.

Alternatively, we can define the capabilities as mappings from **Regvar** to $(0, 1] \cup \{\text{undefined}\}$ and formalize a type system with such kind of capabilities like in other fractional capability calculus [3]. Then, having $\Psi$ such that $\Psi(\rho) = \text{undefined}$ means that the region $\rho$ can be neither deleted nor accessed, $\Psi(\rho) > 0$ means that the region $\rho$ can be accessed, and $\Psi(\rho) = 1$ means that the region $\rho$ can be deleted and accessed. Because this strategy does not improve the expressibility and complicates the inference, we do not adopt it and use the range $[0, \infty)$ instead.

### 3.1 Dynamic Semantics

We define the operational semantics of the region language. The semantics is defined as small step reductions from states to states, where a *state* is a triple $(R, h, e)$ consisting of a *region environment* $R$, a *store* $h$, and a *run-time expression* $e$. A region environment maps region variables to $\{0, 1\}$ where 0 indicates that the region has been deleted, and 1 indicates that the region is alive. A store is a mapping from *locations* $\ell$ to values. We extend values to include integers and locations as follows: $v ::= \cdots \mid n \mid \ell@\rho$.

Here, $\ell@\rho$ is a location $\ell$ allocated in the region $\rho$. The semantics trivially guarantees that for any $\ell$, there is a unique $\rho$ such that $\ell@\rho$ appears as a value in the semantics. Also, the semantics guarantees that if $\ell@\rho$ appears somewhere in the reduction, then $\rho \in \mathit{dom}(R)$ at that point, and that $\ell$ in $\mathit{dom}(h)$. However,

$$(R, h, (\lambda x : \tau.e)\ (v)) \to (R, h, e[v/x]) \quad \textbf{App}$$

$$(R, h, (\Lambda\rho.v)[\rho']) \to (R, h, v[\rho'/\rho]) \quad \textbf{TyApp}$$

$$\frac{R(\rho) = 1 \quad \ell \notin dom(h)}{(R, h, \texttt{ref}\ v@\rho) \to (R, h[\ell \mapsto v], \ell@\rho)}\ \textbf{Ref}$$

$$\frac{\rho \notin dom(R)}{(R, h, \texttt{letreg}\ \rho\ \texttt{in}\ e) \to (R[\rho \mapsto 1], h, e)}\ \textbf{Letreg}$$

$$\frac{R(\rho) = 1}{(R, h, \texttt{free}\ \rho) \to (R[\rho \mapsto 0], h, 0)}\ \textbf{Free}$$

$$\frac{e \to e'}{E[e] \to E[e']}\ \textbf{Context}$$

**Fig. 3.** Representative reduction rules.

it does not guarantee that $R(\rho) = 1$ at such a point, which, importantly, allows dangling pointers, and therefore, memory unsafe programs. It is the job of the static system to reject such unsafe programs (cf. Section 3.2).

Figure 3 shows a few representative reduction rules (see Figure 9 in the appendix for the complete set of rules). **App** handles function calls and **TyApp** handles region parameter passing. Here, the substitution $e[\rho_1/\rho_2]$ is defined in the standard capture-avoiding way (see Figure 10 in the appendix). We defer the definition of substitution for types $\tau[\rho_1/\rho_2]$ to Section 3.2, which is also capture avoiding. We let expressions and types equivalent up to renaming of bound variables and region variables.

**Ref** handles reference allocations, and requires the region where the reference is allocated to be alive. The reduction gets stuck when trying to access a deleted region. The reduction rules for other reference accesses (i.e., reads and writes) are similar (cf. Figure 9). Here, the notation $f[u \mapsto v]$ denotes the extension of $f$ by $u$ mapping to $v$. That is, $f[u \mapsto v] = f \cup \{u \mapsto v\}$ for $u \notin dom(f)$. **Letreg** creates a fresh region, and **Free** frees a live region. Reduction gets stuck when trying to free an already freed region. **Context** is the usual rule to allow reduction in an evaluation context. The evaluation contexts are defined as follows.

$$E ::= [\,] \mid E\ (e) \mid v\ (E) \mid E[\rho] \mid \texttt{ref}\ E@\rho \mid E\ \texttt{:=}\ e \mid v\ \texttt{:=}\ E \mid\ !E$$

Let us write $(R, h, e) \to^* (R', h', e')$ for zero or more reduction steps from the state $(R, h, e)$ to $(R', h', e')$. We say that the program $e$ is memory safe if reducing from the initial state $(\emptyset, \emptyset, e)$ does not get stuck. More formally,

**Definition 1 (Safety).** *We say that $e$ is memory safe if for any state $(R_1, h_1, e_1)$ such that $(\emptyset, \emptyset, e) \to^* (R_1, h_1, e_1)$, either $e_1$ is a value or there exists a state $(R_2, h_2, e_2)$ such that $(R_1, h_1, e_1) \to (R_2, h_2, e_2)$.*

$$\frac{}{\Gamma, \Psi \vdash n : int, \Psi, \emptyset}\ \textbf{\textit{Int}} \qquad \frac{}{\Gamma, \Psi \vdash x : \Gamma(x), \Psi, \emptyset}\ \textbf{\textit{Var}}$$

$$\frac{\Gamma[x \mapsto \tau_{in}], \Psi_{in} \vdash e : \tau_{out}, \Psi_{out}, L \quad \forall \rho \notin L. \Psi_{in}(\rho) = \Psi_{out}(\rho)}{\Gamma, \Psi \vdash \lambda x : \tau_{in}.e : (\Psi_{in}, \tau_{in}) \xrightarrow{L} (\Psi_{out}, \tau_{out}), \Psi, \emptyset}\ \textbf{\textit{Fun}}$$

$$\frac{}{\Gamma, \Psi \vdash \ell@\rho : ref\,(\Gamma(\ell))@\rho, \Psi, \emptyset}\ \textbf{\textit{Loc}} \qquad \frac{\Gamma[x \mapsto \tau], \Psi \vdash v : \tau, \Psi, \emptyset}{\Gamma, \Psi \vdash \texttt{fix}\ x : \tau.v : \tau, \Psi, \emptyset}\ \textbf{\textit{Fix}}$$

$$\frac{\Gamma, \Psi \vdash e_1 : (\Psi_{in}, \tau_{in}) \xrightarrow{L} (\Psi_{out}, \tau_{out}), \Psi_1, L_1 \quad \Gamma, \Psi_1 \vdash e_2 : \tau_{in}, \Psi_{keep} + \Psi_{in}, L_2}{\Gamma, \Psi \vdash e_1\ (e_2) : \tau_{out}, \Psi_{keep} + \Psi_{out}, L_1 \cup L_2 \cup L}\ \textbf{\textit{App}}$$

$$\frac{\Gamma, \Psi \vdash v : \tau, \Psi', \emptyset \quad \rho \notin free(\Gamma)}{\Gamma, \Psi \vdash \Lambda\rho.v : \forall\rho.\tau, \Psi', \emptyset}\ \textbf{\textit{RegAbs}} \qquad \frac{\Gamma, \Psi \vdash e : \forall\rho.\tau, \Psi', L}{\Gamma, \Psi \vdash e[\rho'] : \tau[\rho'/\rho], \Psi', L}\ \textbf{\textit{TyApp}}$$

$$\frac{\Gamma, \Psi \vdash e : \tau, \Psi', L \quad \Psi'(\rho) > 0}{\Gamma, \Psi \vdash \texttt{ref}\ e@\rho : ref(\tau)@\rho, \Psi', L}\ \textbf{\textit{Ref}} \qquad \frac{\Gamma, \Psi \vdash e : ref(\tau)@\rho, \Psi', L \quad \Psi'(\rho) > 0}{\Gamma, \Psi \vdash !e : \tau, \Psi', L}\ \textbf{\textit{Deref}}$$

$$\frac{\Gamma, \Psi \vdash e : ref(\tau)@\rho, \Psi_1, L_1 \quad \Gamma, \Psi_1 \vdash e' : \tau, \Psi', L_2 \quad \Psi'(\rho) > 0}{\Gamma, \Psi \vdash e\ \texttt{:=}\ e' : int, \Psi', L_1 \cup L_2}\ \textbf{\textit{Write}}$$

$$\frac{\Gamma, \Psi + \{\rho \mapsto 1\} \vdash e : \tau, \Psi', L \quad \rho \notin free(\Gamma) \cup free(\tau) \quad \Psi(\rho) = \Psi'(\rho) = 0}{\Gamma, \Psi \vdash \texttt{letreg}\ \rho\ \texttt{in}\ e : \tau, \Psi', L \setminus \{\rho\}}\ \textbf{\textit{Letreg}}$$

$$\frac{}{\Gamma, \Psi + \{\rho \mapsto 1\} \vdash \texttt{free}\ \rho : int, \Psi, \{\rho\}}\ \textbf{\textit{Free}}$$

**Fig. 4.** The type checking rules.

### 3.2 Static Semantics

A capability calculus is a type system, and consists of a set of deductive typing rules. We present the polymorphic fractional capability calculus that guarantees that a typable program is memory safe.

Figure 4 presents the typing rules. The typing judgements are of the form $\Gamma, \Psi \vdash e : \tau, \Psi', L$. Here, the *type environment* $\Gamma$ maps variables to types, *pre-capability* $\Psi$ is the capability of the program before the evaluation of $e$, *post-capability* $\Psi'$ is the capability after evaluating $e$, $\tau$ is the type of $e$, and $L$ is the effect of $e$.

We briefly describe the typing rules. ***Int*** and ***Var*** are self-explanatory. In ***Fun***, we type check the body starting with the pre-capability of the function $\Psi_{in}$ and ending in the post-capability $\Psi_{out}$. We also record $L$ as the latent effect

$$int[\rho_1/\rho_2] = int$$

$$(ref(\tau)@\rho_1')[\rho_1/\rho_2] = ref(\tau[\rho_1/\rho_2])@\rho_1'[\rho_1/\rho_2]$$

$$(\forall\rho_1'.\tau)[\rho_1/\rho_2] = \forall\rho_1'.(\tau[\rho_1/\rho_2]) \quad \text{where } \rho_1' \neq \rho_2$$

$$((\Psi_{in},\tau) \xrightarrow{L} (\Psi_{out},\tau'))[\rho_1/\rho_2] = (\Psi_{in}[\rho_1/\rho_2]^L, \tau[\rho_1/\rho_2]) \xrightarrow{L[\rho_1/\rho_2]} (\Psi_{out}[\rho_1/\rho_2]^L, \tau'[\rho_1/\rho_2])$$

**Fig. 5.** $\tau[\rho_1/\rho_2]$

$$\Psi[\rho_1/\rho_2]^L = \begin{cases} \Psi[\rho_1/\rho_2]_{\Downarrow} & \text{if } \rho_1 \notin L[\rho_1/\rho_2] \\ \Psi[\rho_1/\rho_2]_{\oplus} & \text{if } \rho_1 \in L[\rho_1/\rho_2] \end{cases}$$

**Fig. 6.** $\Psi[\rho_1/\rho_2]^L$ (see Section 2 for the definitions of $\Psi[\rho_1/\rho_2]_{\oplus}$ and $\Psi[\rho_1/\rho_2]_{\Downarrow}$).

of the function. The condition $\forall\rho \notin L.\Psi_{in}(\rho) = \Psi_{out}(\rho)$ becomes handy when proving the soundness of the type system.[4]

***RegAbs***, ***Fix***, and ***Loc*** are self-explanatory. Here, $free(\Gamma)$ is defined to be $\{free(\tau) \mid \tau \in ran(\Gamma)\}$ where $free(\tau)$ is defined as follows.

$$\begin{aligned} &free(int) = \emptyset \\ &free(ref(\tau)@\rho) = free(\tau) \cup \{\rho\} \\ &free((\Psi_{in},\tau_{in}) \xrightarrow{L} (\Psi_{out},\tau_{out})) = free(\tau_{in}) \cup free(\tau_{out}) \cup L \\ &free(\forall\rho.\tau) = free(\tau) \setminus \{\rho\} \end{aligned}$$

***App*** types function applications. The rule takes care of the left-to-write reduction order by connecting the post-capability of $e_1$ (i.e., $\Psi_1$) to the pre-capability of $e_2$. Here the capability addition $\Psi_1 + \Psi_2$ is defined point-wise as $\lambda\rho.\Psi_1(\rho) + \Psi_2(\rho)$. Note that the post-capability of $e_2$, is "split" into $\Psi_{keep}$ and $\Psi_{in}$ so that only $\Psi_{in}$ needs to be given to the function and $\Psi_{keep}$ is kept by the caller and combined with the post-capability of the function. This capability "flow around" technique provides context sensitivity as each call site can use a different $\Psi_{keep}$ to avoid conflating capabilities. (Note, however, that this context sensitivity is orthogonal to parametric region polymorphism.) The flow around technique is inspired by similar ideas used in Cqual [6] and Locksmith [8], and has also been used in the fractional capability calculus for race detection [10].

***TyApp*** handles region instantiation. The type substitution $\tau[\rho'/\rho]$ is non-standard and is defined in Figure 5. The substitution rules for integer types, reference types, and region polymorphic types are self-explanatory (recall that the substitution is capture avoiding). For function types, we instantiate its pre-capability and the post-capability via the special substitution of the form $\Psi[\rho_1/\rho_2]^L$. Figure 6 defines the substitution. Note that it only does the additive instantiation for region variables in $L$, and does the down instantiation for

[4] It is actually redundant for closed programs as it can be derived as a lemma.

other regions. This formally implements the controlled additive instantiation discussed in Section 2. We explain the down instantiation in further detail. The down instantiation is applied for regions that are only accessed but not deleted in the function. Since accessing these regions needs capabilities greater than 0, a function caller needs a capability that satisfies the constraint

$$\Psi_{\text{caller}}(\rho) > 0 \text{ if } \Psi_{\text{pre}}(\rho) > 0$$

where $\Psi_{\text{caller}}$ is the capability necessary to call the function, and $\Psi_{\text{pre}}$ is the pre-capability of the function. However, because linear programming cannot deal with the logical implication, we transform the constraint into the following constraint: [5]

$$\Psi_{\text{caller}}(\rho) \geq \tfrac{1}{2}\Psi_{\text{pre}}(\rho)$$

***Ref***, ***Deref***, and ***Write*** type check reference accesses by checking that the program has enough capability for the accesses. Unlike in the usual effect-based region calculus, the rules do not add the accessed regions to the effect.

***Letreg*** creates a new region and adds the capability to access the region to the pre-capability of $e$. ***Free*** frees a region. Note that the pre-capability is required to have the full capability to access the region and deletes it from the post-capability. ***Free*** also adds $\rho$ to the effect to record that the expression deletes $\rho$.

Next, we define the notion of a well-typed state. Let $0 = \lambda\rho.0$, that is, a capability that maps all regions to 0. We write $\Gamma \vdash h$ if for each $\ell \in dom(h)$, $\Gamma, 0 \vdash h(\ell) : \Gamma(\ell), 0, \emptyset$. We write $\vdash \Gamma$ if for all function types $(\tau, \Psi_{in}) \rightarrow^{L} (\tau', \Psi_{out})$ appearing in $\Gamma$, $\rho \notin L$ implies $\Psi_{in}(\rho) = \Psi_{out}(\rho)$.

**Definition 2 (Well-typed State).** *We write $\Gamma \vdash (R, h, e, \tau, \Psi')$ if there exist $\Psi$ and $L$ such that (1) $\Gamma \vdash h$, (2) $\Gamma, \Psi \vdash e : \tau, \Psi', L$, (3) $R(\rho) \geq \Psi(\rho)$ for all $\rho \in dom(R)$, and (4) $\vdash \Gamma$.*

We prove that typability is preserved under region instantiations, with a "large enough" effect. The proofs appear in the appendix.

**Lemma 1 (Region Variable Substitution).** *Suppose $\Gamma, \Psi \vdash e : \tau, \Psi', L'$ and $L' \subseteq L$. Then, $\Gamma[\rho'/\rho], \Psi[\rho'/\rho]^{L} \vdash e[\rho'/\rho] : \tau[\rho'/\rho], \Psi'[\rho'/\rho]^{L}, L'[\rho'/\rho]$.*

Using the lemma, we show that typability is preserved across reductions.

**Lemma 2 (Preservation).** *Suppose $\Gamma \vdash (R, h, e, \tau, \Psi')$ and $(R, h, e) \rightarrow (R', h', e')$. Then, there exists $\Gamma' \supseteq \Gamma$ such that $\Gamma' \vdash (R', h', e', \tau, \Psi')$.*

**Lemma 3 (Progress).** *Suppose $\Gamma \vdash (R, h, e, \tau, \Psi')$ and $e$ is a closed term. Then, either $e$ is a value or there exist $R'$, $h'$, and $e'$ such that $(R, h, e) \rightarrow (R', h', e')$.*

[5] Dividing capabilities by a number greater than 2 is still sound. For now, we adopt 2, because choosing a greater number does not contribute to the precision of the analysis.

From Lemma 2 and Lemma 3, it follows that if $e$ is well-typed then $e$ is memory safe.

**Theorem 1.** *Suppose* $\emptyset \vdash (\emptyset, \emptyset, e, \tau, \Psi')$. *Then,* $e$ *is memory safe.*

## 4 Capability Inference

We now give a polynomial time capability inference algorithm for the capability calculus. Note that the focus of the paper is not region inference, and therefore, the input program is an expression in the source syntax (cf. Figure 2) that already contains region commands (i.e., region creations, region deletions, and region abstractions). In addition, the algorithm assumes that it is given types for the bound variables, including the effects, except for the capabilities (i.e., $\Psi$'s). Such types may be provided externally as in the original capability calculus [5],or inferred by known techniques [15, 9]. We note that there can be more than one valid region and effect annotations for a program. For example, using [9], polymorphism is restricted to only over `let` and `fix` bound variables (i.e., not $\lambda$-bound variables). But, our algorithm can infer capabilities given any valid annotation, in time polynomial in the size of the annotation.

More formally, we assume that every variable binding in the program is annotated by the following signature $\sigma$.

$$
\begin{array}{l}
\psi \in \textbf{Capvar} \\
\varphi ::= \psi \mid \varphi[\rho'/\rho]^L \\
\sigma ::= \mathit{int} \mid \mathit{ref}(\sigma)@\rho \mid (\varphi_{in}, \sigma_{in}) \xrightarrow{L} (\varphi_{out}, \sigma_{out}) \mid \forall \rho.\sigma
\end{array}
$$

Here, *capability variables* $\psi$ are place holders for the actual capabilities to be inferred by the algorithm (we use the lower-case "psi" for the variables). Each $\varphi$ appearing in the annotation is a distinct capability variable (i.e., some $\psi$). The form $\varphi[\rho'/\rho]^L$ appears during the constraint generation. Unless specified otherwise, we overload $e$ to expressions with $\sigma$ annotations. Without loss of generality, we assume that bound variables and region variables are distinct in the given program.

The inference judgements are of the form $\Delta, \psi \vdash e : \sigma, \psi', L \Rightarrow C$, which is read "given environment $\Delta$, $e$ is inferred to have the signature $\sigma$, the pre-capability variable $\psi$, the post-capability variable $\psi'$, and the effect $L$ with the set of *capability constraints* $C$". A capability constraint is of the following forms.

$$
\begin{array}{ccc}
\psi(\rho) = \psi'(\rho) & \psi(\rho) > 0 & \varphi = \varphi' \\
\psi = \psi' + \{\rho \mapsto 1\} & \psi(\rho) = 0 & \varphi_0 = \varphi_1 + \varphi_2
\end{array}
$$

To generate constraints, we initialize $\Delta$ such that for each variable $x$, $\Delta(x) = \sigma$ where $\sigma$ is the signature of $x$ given in the annotation, that is, either $\texttt{fix}x : \sigma.v$ or $\lambda x : \sigma.e$ appears in the program (recall that bound variables are distinct). We also pick distinct capability variables $\psi_{start}$ and $\psi_{end}$, and we generate constraints $C$ by $\Delta, \psi_{start} \vdash e : \sigma, \psi_{end}, L \Rightarrow C$ where $e$ is the input program, and $\psi_{start}, \psi_{end}$ are distinct capability variables.

$$\frac{\psi \text{ fresh} \quad \Delta, \psi_{in} \vdash e : \sigma_{out}, \psi_{out}, L \Rightarrow C}{\Delta, \psi \vdash (\lambda x : \sigma_{in}.e) : (\psi_{in}, \sigma_{in}) \xrightarrow{L} (\psi_{out}, \sigma_{out}), \psi, \emptyset \Rightarrow C \cup \{\psi_{in}(\rho) = \psi_{out}(\rho) \mid \rho \in \mathfrak{L} \setminus L\}} \textbf{cFun}$$

$$\frac{\psi, \psi' \text{ fresh}}{\Delta, \psi \vdash \texttt{free}\,\rho : int, \psi', \{\rho\} \Rightarrow \{\psi = \psi' + \{\rho \mapsto 1\}\}} \textbf{cFree}$$

$$\frac{\begin{array}{c}\Delta, \psi \vdash e_1 : (\varphi_{in}, \sigma_{in}) \xrightarrow{L} (\varphi_{out}, \sigma_{out}), \psi_1, L_1 \Rightarrow C_1 \\ \Delta, \psi_2 \vdash e_3 : \sigma', \psi_3, L_2 \Rightarrow C_2 \quad \vdash \sigma_{in} =_u \sigma' \Rightarrow C_3 \quad \psi_{keep} \text{ fresh}\end{array}}{\begin{array}{l}\Delta, \psi \vdash e_1\ (e_2) : \sigma_{out}, \psi', L_1 \cup L_2 \cup L \Rightarrow \\ \qquad C_1 \cup C_2 \cup C_3 \cup \{\psi_1 = \psi_2, \psi_3 = \varphi_{in} + \psi_{keep}, \psi' = \psi_{keep} + \varphi_{out}\}\end{array}} \textbf{cApp}$$

**Fig. 7.** Representative constraint generation rules.

Figure 7 shows a few representative constraint generation rules (see Figure 11 in the appendix for the complete set of rules). Each rule is a straightforward syntax-directed constraint generation rule for the type checking rules from Figure 4. The rules use the relation $\vdash \sigma =_u \sigma' \Rightarrow C$ to generate capability equality constraints of the form $\varphi = \varphi'$. The relation is defined inductively on the structure of the types, and is straightforward (see Figure 12 in the appendix). We assume that the type annotations in the input program are correct in the sense that the instances of the rules are all well-defined in the constraint generation.

The substitution $\sigma[\rho'/\rho]$ is defined exactly like $\tau[\rho'/\rho]$ from Figure 5 (i.e., inductively on the structure of $\sigma$) with the substitution $\varphi[\rho'/\rho]^L$ just interpreted syntactically.

Let $\mathfrak{L}$ be the set of regions occurring in the program. For each capability variable $\psi$ and a region $\rho \in \mathfrak{L}$, we associate a distinct linear programming variable $\xi_{\psi,\rho}$ to denote $\psi(\rho)$. We reduce the capability constraints to linear inequality constraints by applying the following rules.

$$\begin{array}{lcl}
\psi(\rho) = \psi'(\rho) & \Rightarrow & \{\xi_{\psi,\rho} = \xi_{\psi',\rho}\} \\
\psi(\rho) > 0 & \Rightarrow & \{\xi_{\psi,\rho} > 0\} \\
\varphi = \varphi' & \Rightarrow & \{S(\varphi, \rho) = S(\varphi', \rho) \mid \rho \in \mathfrak{L}\} \\
\psi = \psi' + \{\rho_f \mapsto 1\} & \Rightarrow & \{\xi_{\psi,\rho} = \xi_{\psi',\rho} \mid \rho \in \mathfrak{L} \setminus \{\rho_f\}\} \cup \{\xi_{\psi,\rho_f} = \xi_{\psi',\rho_f} + 1\} \\
\psi(\rho) = 0 & \Rightarrow & \{\xi_{\psi,\rho} = 0\} \\
\varphi_0 = \varphi_1 + \varphi_2 & \Rightarrow & \{S(\varphi_0, \rho) = S(\varphi_1, \rho) + S(\varphi_2, \rho) \mid \rho \in \mathfrak{L}\}
\end{array}$$

where $S(\varphi, \rho)$ is defined as follows.

$$\begin{array}{ll}
S(\psi, \rho) & = \xi_{\psi,\rho} \\
S(\varphi[\rho_a/\rho_x]^L, \rho) & = \text{if } \rho = \rho_x \text{ then } 0 \\
& \quad \text{else if } \rho \neq \rho_a \text{ then } S(\varphi, \rho) \\
& \quad \text{else if } \rho_x \notin L \text{ then } \tfrac{1}{2}(S(\varphi, \rho_x) + S(\varphi, \rho_a)) \\
& \quad \text{else } S(\varphi, \rho_x) + S(\varphi, \rho_a)
\end{array}$$

We also add the inequality $\xi_{\psi,\rho} \geq 0$ for each $\xi_{\psi,\rho}$ to ensure that capabilities are within the range $[0, \infty)$. Then, we check whether there exists a solution to the set of these inequalities, and if so, we accept the program as safe, and otherwise, we reject the program.

To apply linear programing algorithms that can only take non-strict inequalities such as GLPK [1], we add a fresh linear programming variable $\xi_s$ and replace each $\xi_{\psi,\rho} > 0$ with $\xi_{\psi,\rho} \geq \xi_s$, and set the objective function to be $\xi_s$. We then ask the linear programming solver to find a solution that maximizes $\xi_s$ and accept if and only if the solver returns a solution with $\xi_s > 0$.

The soundness and the completeness of the inference is stated and proven in the appendix (cf. Theorems 2 and 3).

### 4.1 Time Complexity

We discuss the time complexity of the capability inference algorithm. Recall that we assume that region and effect annotations are provided.[6] The complexity of the constraint generation is polynomial in the size of the given region and effect annotations, and so is the size of the generated set of linear inequalities. The complexity of (rational) linear programming is polynomial in the size of the linear inequalities. Therefore, the complexity of our inference algorithm is polynomial in the size of the provided types.

*NP-hardness of non-fractional capability calculus:* It is possible to show that restricting capabilities to range only over the set $\{0, 1\}$ instead of the range $[0, \infty)$ renders the capability inference NP-hard, even without polymorphism [13, 7]. Therefore, not only is the fractional capability calculus able to prove more programs memory safe than the non-fractional variant, it is actually more computationally tractable, assuming that P $\neq$ NP.

## 5 Adding Polymorphism to Other Fractional Capability Calculi

Our approach can be used to add parametric polymorphism to other fractional capability calculus applications. We take race detection [10] for instance and discuss the methodology.

Recall that, for the region calculus, we may use the down instantiation for regions that are only accessed (i.e., read, written, or allocated) but not deleted in the function. Because the race detection calculus needs to distinguish reads from writes[7], we use the down instantiation for abstract locations[8] that are only

[6] They may be inferred in time polynomial in the size of the underlying Hindley-Milner type under certain restrictions (i.e., impredicativity) via methods like [9].

[7] Formally, a race occurs when two accesses to a memory location happens concurrently such that one of them is a write, and so a concurrent read is not a race.

[8] Abstract locations soundly approximates the actual locations. They serve the role similar to region variables in the region calculus, and we overload $\rho$ to range over abstract locations in this section.

read but not written. In the race calculus, writes and reads are distinguished by requiring the capability greater or equal to 1 for writes and capability greater than 0 for reads. We define a new instantiation scheme, called *1-instantiation*, to instantiate abstract locations that may be written in functions:

$$\Psi[\rho_1/\rho_2]_1 = \{\rho_1 \mapsto 1\} \cup \{\rho \mapsto \Psi(\rho) \mid \rho \notin \{\rho_1, \rho_2\}\}$$

Note that this scheme lowers the capability for $\rho_2$ to 1. We use effects to track the abstract locations that may be written.

To prove Lemma 1, it is important that the effect $L$ contains the regions where pre-capability (i.e., $\Psi$) and post-capability (i.e., $\Psi'$) are different (cf. Appendix A). For the region calculus, this occurs only when the ***Free*** type rule is applied (see Figure 4). In the race detection calculus, this occurs when capabilities are passed across threads via operations like lock creation, lock acquisition, and lock release. For example, the following rule is applied for lock creation.

$$\frac{}{\Gamma, \Psi + \Psi_1 \vdash \mathsf{newlock} : \mathrm{lock}(\Psi_1), \Psi}$$

Note that pre-capability, $\Psi + \Psi_1$, may differ from the post-capability, $\Psi$, and so we need to track the abstract locations where the capabilities differ as effects. Like in the region calculus, we use additive instantiation for such abstraction locations.

To summarize, we have three instantiation schemes for the race detection calculus: the down instantiation for read-only locations, 1-instantiation for read-or-write-only locations, and the additive instantiation for the rest (i.e., ones that may be possibly passed across threads).

As in the region calculus, we use effects to select the instantiation schemes. We now have two kinds of effects, one for abstract locations that change their capabilities ($L_1$ in the definition below), and one for abstract locations that are written ($L_2$ in the definition below). The new instantiation rule is defined as follows.

$$\Psi[\rho_1/\rho_2]^{L_1,L_2} = \begin{cases} \Psi[\rho_1/\rho_2]_{\Downarrow} & \text{if } \rho_1 \notin (L_1 \cup L_2)[\rho_1/\rho_2] \\ \Psi[\rho_1/\rho_2]_{\oplus} & \text{if } \rho_1 \in L_1[\rho_1/\rho_2] \\ \Psi[\rho_1/\rho_2]_1 & \text{if } \rho_1 \in (L_2 \setminus L_1)[\rho_1/\rho_2] \end{cases}$$

We modify the type checking rules to track effects. Judgements are of the form $\Gamma, \Psi \vdash e : \tau, \Psi', L_1, L_2$ where $L_1$ overapproximates abstract locations that may change in $e$, and $L_2$ overapproximates abstract locations that are written in $e$.

Figure 8 shows a few representative type checking rules. **WRITE** rule adds the written abstract location in the effect $L_2$. In **NEWL** rule, we add the abstract locations that are passed by the lock creation, that is, $\mathit{positive}(\Psi_1)$. Inferring $\mathit{positive}(\Psi_1)$ requires solving linear inequalities, inducing cyclic dependencies between capability constraint generation and capability constraint solving. Fortunately, it is sound to overapproximate effects, and so a tractable approach is to use all abstract locations (i.e., $\mathfrak{L}$) in place of $\mathit{positive}(\Psi_1)$ at **NEWL** (and other type rules that also change capabilities). By an argument analogous to the one in Section 4.1, it can be shown that the capability inference for such a system is polynomial time computable.

$$\frac{\Gamma, \Psi \vdash e_1 : \mathit{ref}(\tau)@\rho, \Psi_1, L_1, L_2 \quad \Gamma, \Psi_1 \vdash e_2 : \tau, \Psi_2, L_1', L_2' \quad \Psi_2(\rho) \geq 1}{\Gamma, \Psi \vdash e_1 := e_2 : \mathit{int}, \Psi_2, L_1 \cup L_1', L_2 \cup L_2' \cup \{\rho\}} \textbf{ WRITE}$$

$$\frac{}{\Gamma, \Psi + \Psi_1 \vdash \mathsf{newlock} : \mathrm{lock}(\Psi_1), \Psi, \mathit{positive}(\Psi_1), \emptyset} \textbf{ NEWL}$$

where $\mathit{positive}(\Psi) = \{\rho \in \mathit{dom}(\Psi) \mid \Psi(\rho) > 0\}$

**Fig. 8.** Representative polymorphic fractional race typing rules

## 6 Related Work and Conclusion

Fractional capabilities were originally proposed by Boyland to guarantee determinism of multi-thread programs while permitting parallel reads [4]. For the monomorphic fragment, it has been shown that the type inference can be solved efficiently by a reduction to linear programming [12, 13], and later work has exploited this observation to create efficient fractional-capability-based program analyses, ranging from race detection to security analyses [7, 14, 10, 11].

While extending these calculi to parametric polymorphism is discussed in some of the papers (e.g., [4, 13, 10]), none has shown how to do type inference efficiently in the presence of polymorphism. This paper addresses the issue by presenting a general methodology to extend a fractional capability calculus to parametric polymorphism while preserving soundness and the ability to do efficient type inference.

## References


1. GNU Linear Programming Kit. *http:/www.gnu.org/software/glpk/glpk.html.*
2. Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 301–320, 2007.
3. Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 259–270, 2005.
4. John Boyland. Checking interference with fractional permissions. In *Static Analysis, 10th International Symposium, SAS*, pages 55–72. 2003.
5. Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 262–275, 1999.
6. Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 1–12, 2002.

7. Daisuke Kikuchi and Naoki Kobayashi. Type-based verification of correspondence assertions for communication protocols. In *Programming Languages and Systems, 5th Asian Symposium, APLAS*, pages 191–205. 2007.
8. Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 320–331, 2006.
9. Jakob Rehof and Manuel Fähndrich. Type-base flow analysis: from polymorphic subtyping to cfl-reachability. In *Proceedings of the 23th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 54–66, 2001.
10. Tachio Terauchi. Checking race freedom via linear programming. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 1–10, 2008.
11. Tachio Terauchi. A type system for observational determinism. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF*, pages 287–300, 2008.
12. Tachio Terauchi and Alex Aiken. Witnessing side-effects. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP*, pages 105–115, 2005.
13. Tachio Terauchi and Alex Aiken. A capability calculus for concurrency and determinism. *ACM Trans. Program. Lang. Syst.*, 2008.
14. Tachio Terauchi and Adam Megacz. Inferring channel buffer bounds via linear programming. In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP*, pages 284–298. 2008.
15. Mads Tofte and Jean P. Talpin. Implementation of the typed call-by-value $\lambda$-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 188–201, 1994.

## A Omitted Proofs

**Definition 3.** *We write* $\Psi \geq \Psi'$ *if for all* $\rho$, $\Psi(\rho) \geq \Psi'(\rho)$.

**Definition 4.** $\Psi + \Psi' = \lambda\rho.\Psi(\rho) + \Psi'(\rho)$

**Definition 5.** $\mathit{positive}(\Psi) = \{\rho \mid \Psi(\rho) > 0\}$

**Definition 6.** *We write* $\vdash \Gamma$ *if for all function types* $(\tau, \Psi_{in}) \rightarrow^L (\tau', \Psi_{out})$ *in* $\Gamma$, $\forall \rho \notin L.\Psi_{in} = \Psi_{out}$.

**Lemma 4.** *Suppose* $\Psi \geq \Psi'$. *Then,* $\Psi[\rho'/\rho]^L \geq \Psi'[\rho'/\rho]^L$.

*Proof.* Trivial.

**Lemma 5.** *Suppose* $\Gamma, \Psi \vdash e : \tau, \Psi', L$. *Then,* $\Gamma, \Psi + \Psi^+ \vdash e : \tau, \Psi' + \Psi^+, L$

*Proof.* We prove by induction on typing derivations. We only show the ***L*etreg** case. Other cases are trivial.

By induction hypothesis, we have $\Gamma, \Psi + \{\rho \mapsto 1\} + \Psi^+ \vdash e : \tau, \Psi' + \Psi^+, L$. By $\alpha$-conversion, we have $\rho \notin \mathsf{positive}(\Psi^+)$. Therefore, $\Psi(\rho) = \Psi'(\rho) = 0$. Thus, $\Gamma, \Psi + \Psi^+ \vdash \mathtt{letreg}\,\rho\,\mathtt{in}\,e : \tau, \Psi' + \Psi^+, L \setminus \{\rho\}$.

**Lemma 6.** *There exist $\Gamma$ and $\tau$ such that $\Gamma, \Psi \vdash v : \tau, \Psi, \emptyset$*

*Proof.* Trivial.

**Lemma 7.** *Suppose $\vdash \Gamma$ and $\Gamma, \Psi \vdash e : \tau, \Psi', L'$. Then, for all function types $(\tau, \Psi_{in}) \rightarrow^{L} (\tau', \Psi_{out})$ in $\tau$, $\forall \rho' \notin L. \Psi_{in}(\rho') = \Psi_{out}(\rho')$.*

*Proof.* Trivial.

**Lemma 1.** *Suppose the following conditions.*

- $\Gamma, \Psi \vdash e : \tau, \Psi', L'$
- $\vdash \Gamma$
- $L' \subseteq L$

*Then, $\Gamma[\rho'/\rho], \Psi[\rho'/\rho]^{L} \vdash e[\rho'/\rho] : \tau[\rho'/\rho], \Psi'[\rho'/\rho]^{L}, L'[\rho'/\rho]$*

*Proof.* We prove by induction on derivation of type.

- ***Int***
- ***Var***
- ***Loc***
  Trivial.
- ***Fun***
  By I.H.(induction hypothesis), We have

  $$(\Gamma, x : \tau)[\rho'/\rho], \Psi_{\text{in}}[\rho'/\rho]^{L'[\rho'/\rho]} \vdash e[\rho'/\rho] : \tau'[\rho'/\rho], \Psi_{\text{out}}[\rho'/\rho]^{L'[\rho'/\rho]}, L'[\rho'/\rho]$$

  Then, we have

  $$\Gamma[\rho'/\rho], \Psi[\rho'/\rho]^{\emptyset} \vdash (\lambda x : \tau.e)[\rho'/\rho] : (\tau, \Psi_{\text{in}}) \rightarrow^{L'[\rho'/\rho]} (\tau', \Psi_{\text{out}})[\rho'/\rho], \Psi[\rho'/\rho]^{\emptyset}, \emptyset$$

- ***App***
  By the assumptions, we have the following properties
  1. $\Gamma, \Psi \vdash e : (\tau, \Psi_{\text{in}}) \rightarrow^{L} (\tau', \Psi_{\text{out}}), \Psi_0, L_0$
  2. $\Gamma, \Psi_0 \vdash e' : \tau, \Psi_{\text{keep}} + \Psi_{\text{in}}, L_1$
  3. $\Gamma, \Psi \vdash e\ (e') : \tau', \Psi_{\text{keep}} + \Psi_{\text{out}}, L_0 \cup L_1 \cup L$
  4. $L_0 \cup L_1 \cup L \subseteq L_s$

  By I.H. and property 1, we have

  $$\begin{aligned}&\Gamma[\rho'/\rho], \Psi[\rho'/\rho]^{L_s} \vdash e[\rho'/\rho] : ((\tau, \Psi_{\text{in}}) \rightarrow^{L} (\tau', \Psi_{\text{out}}))[\rho'/\rho], \Psi_0[\rho'/\rho]^{L_s}, L_0[\rho'/\rho] \\ &\equiv \Gamma[\rho'/\rho], \Psi[\rho'/\rho]^{L_s} \vdash e[\rho'/\rho] : \\ &\qquad (\tau[\rho'/\rho], \Psi_{\text{in}}[\rho'/\rho]^{L}) \rightarrow^{L[\rho'/\rho]} (\tau'[\rho'/\rho], \Psi_{\text{out}}[\rho'/\rho]^{L}), \Psi_0[\rho'/\rho]^{L_s}, L_0[\rho'/\rho]\end{aligned}$$

  By I.H. and property 2, we have

  $$\Gamma[\rho'/\rho], \Psi_0[\rho'/\rho]^{L_s} \vdash e'[\rho'/\rho] : \tau[\rho'/\rho], (\Psi_{\text{in}} + \Psi_{\text{keep}})[\rho'/\rho]^{L_s}, L_1[\rho'/\rho]$$

  We now show that $\rho' \notin L[\rho'/\rho] \wedge \forall \rho_x \notin L. \Psi(\rho_x) = \Psi'(\rho_x) \Rightarrow \Psi(\rho') = \Psi'(\rho') \wedge \Psi(\rho) = \Psi'(\rho)$. If $\rho' \in L$, then $\rho' \in L[\rho'/\rho]$. It leads to a contradiction.

Therefore, $\rho' \notin L$. Hence, $\Psi(\rho') = \Psi'(\rho')$. By the same fashion, we have $\rho \notin L$. Hence, $\Psi(\rho) = \Psi'(\rho)$.
Here, we show $\Psi_{\text{out}}[\rho'/\rho]^L + \Psi'_{\text{keep}} = (\Psi_{\text{out}} + \Psi_{\text{keep}})[\rho'/\rho]^{L_s}$ by case analysis.
By $\vdash \Gamma$, we have $\rho \notin L.\Psi_{\text{in}}(\rho) = \Psi_{\text{out}}(\rho)$. Let $X(\rho_x)$ and $Y(\rho_y)$ be as follows.

$$\begin{aligned} X(\rho_x) &= \Psi_{\text{out}}[\rho'/\rho]^L(\rho_x) + \Psi'_{\text{keep}}(\rho_x) \\ &= \Psi_{\text{out}}[\rho'/\rho]^L(\rho_x) + (\Psi_{\text{keep}} + \Psi_{\text{in}})[\rho'/\rho]^{L_s}(\rho_x) - \Psi_{\text{in}}[\rho'/\rho]^L(\rho_x) \\ Y(\rho_x) &= (\Psi_{\text{out}} + \Psi_{\text{keep}})[\rho'/\rho]^{L_s}(\rho_x) \end{aligned}$$

It suffices to show that $X(\rho_x) = Y(\rho_y)$ for any cases.
Suppose $\rho_x \neq \rho'$.

$$\begin{aligned} X(\rho_x) &= \Psi_{\text{out}}(\rho_x) + \Psi_{\text{keep}}(\rho_x) + \Psi_{\text{in}}(\rho_x) - \Psi_{\text{in}}(\rho_x) \\ &= \Psi_{\text{out}}(\rho_x) + \Psi_{\text{keep}}(\rho_x) \\ Y(\rho_x) &= \Psi_{\text{out}}(\rho_x) + \Psi_{\text{keep}}(\rho_x) \end{aligned}$$

Suppose $\rho' \in L[\rho'/\rho] \wedge \rho' = \rho_x$.

$$\begin{aligned} X(\rho_x) &= \Psi_{\text{out}}(\rho) + \Psi_{\text{out}}(\rho') + \Psi_{\text{keep}}(\rho) + \Psi_{\text{keep}}(\rho') + \Psi_{\text{in}}(\rho) + \Psi_{\text{in}}(\rho') \\ &\quad -(\Psi_{\text{in}}(\rho) + \Psi_{\text{in}}(\rho')) \\ &= \Psi_{\text{out}}(\rho) + \Psi_{\text{out}}(\rho') + (\Psi_{\text{keep}}(\rho) + \Psi_{\text{keep}}(\rho')) \\ &= Y(\rho_x) \end{aligned}$$

Suppose $\rho' \notin L[\rho'/\rho]$, $\rho' \in L_s[\rho'/\rho]$, and $\rho_x = \rho'$.

$$\begin{aligned} X(\rho_x) &= 1/2(\Psi_{\text{out}}(\rho) + \Psi_{\text{out}}(\rho')) + \Psi_{\text{keep}}(\rho) + \Psi_{\text{keep}}(\rho') + \Psi_{\text{in}}(\rho) + \Psi_{\text{in}}(\rho') \\ &\quad -1/2(\Psi_{\text{in}}(\rho) + \Psi_{\text{in}}(\rho')) \\ &= \Psi_{\text{keep}}(\rho) + \Psi_{\text{keep}}(\rho') + \Psi_{\text{in}}(\rho) + \Psi_{\text{in}}(\rho') \\ &= \Psi_{\text{keep}}(\rho) + \Psi_{\text{keep}}(\rho') + \Psi_{\text{out}}(\rho) + \Psi_{\text{out}}(\rho') \\ &= Y(\rho_x) \end{aligned}$$

Suppose $\rho' \notin L[\rho'/\rho]$, $\rho' \notin L_s[\rho'/\rho]$ and $\rho_x = \rho'$.

$$\begin{aligned} X(\rho_x) &= 1/2(\Psi_{\text{out}}(\rho) + \Psi_{\text{out}}(\rho')) + 1/2(\Psi_{\text{keep}}(\rho) + \Psi_{\text{keep}}(\rho')) + 1/2(\Psi_{\text{in}}(\rho) + \Psi_{\text{in}}(\rho')) \\ &\quad -1/2(\Psi_{\text{in}}(\rho) + \Psi_{\text{in}}(\rho')) \\ &= 1/2(\Psi_{\text{out}}(\rho) + \Psi_{\text{out}}(\rho')) + 1/2(\Psi_{\text{keep}}(\rho) + \Psi_{\text{keep}}(\rho')) \\ &= Y(\rho_x) \end{aligned}$$

Hence, we have $\Gamma[\rho'/\rho], \Psi[\rho'/\rho]^{L_s} \vdash (e(e'))[\rho'/\rho] : \tau[\rho'/\rho], (\Psi_{\text{out}} + \Psi_{\text{keep}})[\rho'/\rho]^{L_s}, (L_0 \cup L_1 \cup L)[\rho'/\rho]$.

- ***Ref***

By I.H., we have

$$\Gamma[\rho'/\rho], \Psi[\rho'/\rho]^{L_s} \vdash e[\rho'/\rho] : \tau[\rho'/\rho], \Psi'[\rho'/\rho]^{L_s}, L[\rho'/\rho]$$

By the premise of the typing derivation, we have $\Psi'(\rho_x) > 0$. Therefore, we have $\Psi'[\rho'/\rho]^{L_s}(\rho_x[\rho'/\rho]) > 0$. Hence, we have

$$\Gamma[\rho'/\rho], \Psi[\rho'/\rho]^{L_s} \vdash (\texttt{ref}\ e@\rho_x)[\rho'/\rho] : (ref(\tau)@\rho_x)[\rho'/\rho], \Psi'[\rho'/\rho]^{L_s}, L[\rho'/\rho]$$

– ***W*rite**
By I.H., we have
- $\Gamma[\rho'/\rho], \Psi[\rho'/\rho]^{L_s} \vdash e[\rho'/\rho] : ref(\tau)@\rho_x[\rho'/\rho], \Psi'[\rho'/\rho]^{L_s}, L[\rho'/\rho]$
- $\Gamma[\rho'/\rho], \Psi'[\rho'/\rho]^{L_s} \vdash e'[\rho'/\rho] : \tau[\rho'/\rho], \Psi''[\rho'/\rho]^{L_s}, L'[\rho'/\rho]$
- $\Psi''[\rho'/\rho]^{L_s}(\rho_x[\rho'/\rho]) > 0$

Then, we have

$$\Gamma[\rho'/\rho], \Psi[\rho'/\rho]^{L_s} \vdash e := e'[\rho'/\rho] : int[\rho'/\rho], \Psi''[\rho'/\rho]^{L_s}, (L \cup L')[\rho'/\rho]$$

– ***R*ead**
By I.H., we have
- $\Gamma[\rho'/\rho], \Psi[\rho'/\rho]^{L_s} \vdash e : ref(\tau)@\rho_x[\rho'/\rho], \Psi'[\rho'/\rho]^{L_s}, L[\rho'/\rho]$
- $\Psi'[\rho'/\rho]^{L_s}(\rho_x[\rho'/\rho]) > 0$

Then, we have

$$\Gamma[\rho'/\rho], \Psi[\rho'/\rho]^{L_s} \vdash !e[\rho'/\rho] : \tau[\rho'/\rho], \Psi'[\rho'/\rho]^{L_s}, L[\rho'/\rho]$$

– ***F*ree**
We have the following derivation.

$$\frac{\Psi[\rho'/\rho]^{L_s} = \Psi'[\rho'/\rho]^{L_s} + \{\rho_x[\rho'/\rho] \mapsto 1\}}{\Gamma[\rho'/\rho], \Psi[\rho'/\rho]^{L_s} \vdash e[\rho'/\rho] : int[\rho'/\rho], \Psi'[\rho'/\rho]^{L_s}, \{\rho_x[\rho'/\rho]\}}\ \textbf{\textit{F}ree}$$

– ***L*etreg**
Because $[\rho'/\rho]$ doesn't contain $\rho_x$ (by $\alpha$ conversion), by I.H., we have $\Gamma[\rho'/\rho], \Psi[\rho'/\rho]^{L_s} + \{\rho_x \mapsto 1\} \vdash e[\rho'/\rho] : \tau[\rho'/\rho], \Psi'[\rho'/\rho]^{L_s}, L[\rho'/\rho]$. By the same manner, we have $\rho_x \notin \mathrm{free}(\Gamma[\rho'/\rho]) \cup \mathrm{free}(\tau[\rho'/\rho])$ and $\Psi[\rho'/\rho]^{L_s}(\rho_x) = \Psi'[\rho'/\rho]^{L_s}(\rho_x) = 0$. Therefore, we have

$$\Gamma[\rho'/\rho], \Psi[\rho'/\rho]^{L_s} \vdash (\texttt{letreg}\,\rho_x\,\texttt{in}\,e)[\rho'/\rho] : \tau[\rho'/\rho], \Psi'[\rho'/\rho]^{L_s}, L[\rho'/\rho]$$

– ***T*yApp**
Because $\rho$ is a fresh region variable ($\alpha$ conversion), we have $[\rho'_x/\rho_x][\rho'/\rho] = [\rho'/\rho][\rho'_x[\rho'/\rho]/\rho_x]$. By I.H., we have the following derivation.

$$\frac{\Gamma[\rho'/\rho], \Psi[\rho'/\rho]^{L_s} \vdash e[\rho'_x][\rho'/\rho] : \forall\rho_x.\tau[\rho'/\rho], \Psi'[\rho'/\rho]^{L_s}, L_s[\rho'/\rho] \qquad [\rho'/\rho]' = [\rho'_x[\rho'/\rho]/\rho_x]}{\Gamma[\rho'/\rho], \Psi[\rho'/\rho]^{L_s} \vdash e[\rho'/\rho][\rho'/\rho]' : \tau[\rho'/\rho][\rho'/\rho]', \Psi'[\rho'/\rho]^{L_s}, L_s[\rho'/\rho]}$$

Therefore, we have $\Gamma[\rho'/\rho], \Psi[\rho'/\rho]^{L_s} \vdash e[\rho'_x/\rho_x][\rho'/\rho] : \tau[\rho'_x/\rho_x][\rho'/\rho], \Psi'[\rho'/\rho]^{L_s}, L_s[\rho'/\rho]$

– ***R*egAbs**
We can suppose $\rho_x \neq \rho$, because of $\alpha$ conversion. By I.H., we have $\Gamma[\rho'/\rho], \Psi[\rho'/\rho] \vdash v[\rho'/\rho] : \tau[\rho'/\rho], \Psi'[\rho'/\rho], \emptyset$. Therefore, we have $\Gamma[\rho'/\rho], \Psi[\rho'/\rho] \vdash \Lambda\rho_x.v[\rho'/\rho] : \forall\rho_x.\tau[\rho'/\rho], \Psi'[\rho'/\rho], \emptyset$.

– ***F*ix**
By I.H., we have $\Gamma[x \mapsto \tau][\rho'/\rho], \Psi[\rho'/\rho] \vdash v[\rho'/\rho] : \tau[\rho'/\rho], \Psi'[\rho'/\rho], \emptyset$. Therefore, we have $\Gamma[\rho'/\rho], \Psi[\rho'/\rho] \vdash \texttt{fix}\,x : \tau.v[\rho'/\rho] : \tau[\rho'/\rho], \Psi'[\rho'/\rho], \emptyset$

**Lemma 8.** *Suppose the following conditions.*

1. $\Gamma, x : \tau, \Psi_{in} \vdash e : \tau', \Psi_{out}, L$
2. $\Gamma, \Psi \vdash v : \tau, \Psi, \emptyset$

*Then,* $\Gamma, \Psi_{in} \vdash e[v/x] : \tau', \Psi_{out}, L$.

*Proof.* By induction on derivation, trivial.

**Definition 7.** *We write* $\Psi \vdash R$ *if for all* $\rho \in dom(R).R(\rho) \geq \Psi(\rho)$.

**Definition 8.** *We write* $\Gamma \vdash (R, h, e, \tau, \Psi')$ *if there exist* $\Psi$ *and* $L$ *such that*

1. $\Gamma, \Psi \vdash e : \tau, \Psi', L$
2. $\Gamma \vdash h$
3. $\Psi \vdash R$
4. $\vdash \Gamma$

**Lemma 9.** *Suppose* $\Gamma, \Psi \vdash e : \tau, \Psi', L$ *and* $\Gamma \subseteq \Gamma'$. *Then,* $\Gamma', \Psi \vdash e : \tau, \Psi', L$.

This lemma is a standard weakening property.

*Proof.* Trivial.

**Lemma 10.** *Suppose* $\Gamma, x : \tau, \Psi \vdash v : \tau, \Psi, \emptyset$. *Then,* $\Gamma, \Psi \vdash v[\mathit{fix}\ x : \tau.v/x] : \tau, \Psi, \emptyset$.

*Proof.* Trivial.

**Lemma 2.** *Suppose* $\Gamma \vdash (R, h, e, \tau, \Psi')$ *and* $(R, h, e) \to (R', h', e')$. *Then,* $\Gamma' \vdash (R', h', e', \tau, \Psi')$ *where* $\Gamma \subseteq \Gamma'$.

*Proof.* By case analysis on evaluation.

- **Ref**
  By assumption, there exists $\Psi$ such that
  1. $\Gamma, \Psi \vdash \texttt{ref}\ v@\rho : \mathit{ref}(\tau)@\rho, \Psi', \emptyset$
  2. $\Gamma \vdash h$
  3. $\Psi \vdash R$
  4. $\vdash \Gamma$
  5. $(R, h, \texttt{ref}\ v@\rho) \to (R, h[\ell \mapsto v], \ell@\rho)$

  $\Psi = \Psi'$ is obvious by Lemma 6. Let $\Gamma'$ be $\Gamma, \ell : \mathit{ref}(\tau)@\rho$, we have

  $$\frac{\Gamma', \Psi \vdash v : \tau, \Psi, \emptyset}{\Gamma', \Psi \vdash \texttt{ref}\ v@\rho : \Gamma'(\ell), \Psi, \emptyset}$$

  If $\tau$ is a function type, we have $\forall \rho' \notin L.\Psi_{\text{in}}(\rho') = \Psi_{\text{out}}(\rho')$ by lemma 7.
  1. $\Gamma', \Psi' \vdash \ell@\rho : \mathit{ref}(\tau)@\rho, \Psi', \emptyset$
  2. $\Gamma' \vdash h[\ell \mapsto v]$
  3. $\Psi' \vdash R$
  4. $\vdash \Gamma'$

  Therefore, we have $\Gamma \vdash (R, h[\ell \mapsto v], \ell@\rho, \mathit{ref}(\tau)@\rho, \Psi')$.

– **Write**
By assumption, there exists $\Psi$ such that
1. $\Gamma, \Psi \vdash \ell @ \rho := v : int, \Psi', \emptyset$
2. $\Gamma \vdash h$
3. $\Psi \vdash R$
4. $\vdash \Gamma$
5. $(R, h, \ell @ \rho := v) \to (R, h[\ell \mapsto v], 0)$

$\Psi = \Psi'$ is obvious by Lemma 6. Because the $h(\ell)$'s type and the $v$'s type are same, we have $\Gamma \vdash h[\ell \mapsto v]$.

1. $\Gamma, \Psi' \vdash 0 : int, \Psi', \emptyset$
2. $\Gamma \vdash h[\ell \mapsto v]$
3. $\Psi \vdash R$
4. $\vdash \Gamma$

Therefore, we have $\Gamma \vdash (R, h[\ell \mapsto v], 0, int, \Psi')$.

– **Read**
1. $\Gamma, \Psi \vdash !\ell @ \rho : \tau, \Psi', \emptyset$
2. $\Gamma \vdash h$
3. $\Psi \vdash R$
4. $\vdash \Gamma$
5. $(R, h, !\ell @ \rho) \to (R, h, h(\ell))$

$\Psi = \Psi'$ is obvious by Lemma 6. By property 2, we have $\Gamma, \Psi' \vdash h(\ell) : \tau, \Psi', \emptyset$.

1. $\Gamma, \Psi' \vdash h(\ell) : \tau, \Psi', \emptyset$
2. $\Gamma \vdash h$
3. $\Psi \vdash R$
4. $\vdash \Gamma$

Therefore, we have $\Gamma \vdash (R, h, h(\ell), \tau, \Psi')$.

– **Free**
By assumption, we have
- $\Gamma, \Psi \vdash \texttt{free}\ \rho : int, \Psi', \{\rho\}$
- $\Gamma \vdash h$
- $\Psi \vdash R$
- $\vdash \Gamma$
- $(R, h, \texttt{free}\ \rho) \to (R[\rho \mapsto 0], h, 0)$

and by premise of ***Free*** rule, $\Psi = \Psi' + \{\rho \mapsto 1\}$. By property 4, we have $\Psi'(\rho) = 0$. Then we have $\Gamma \vdash h$ and $\Psi' \vdash R[\rho \mapsto 0]$. Therefore, we have $\Gamma \vdash (R[\rho \mapsto 0], h, 0, int, \Psi')$.

– **Letreg**
By assumption, we have
- $\Gamma, \Psi \vdash \texttt{letreg}\ \rho\ \texttt{in}\ e : \tau, \Psi', L$
- $\Gamma \vdash h$
- $\Psi \vdash R$
- $\vdash \Gamma$
- $(R, h, \texttt{letreg}\ \rho\ \texttt{in}\ e) \to (R[\rho \mapsto 1], h, e)$

By the premise of type derivation, we have

$$\Gamma, \Psi + \{\rho \mapsto 1\} \vdash e : \tau, \Psi', L$$

By premise of property 1, we have $\Gamma, \Psi + \{\rho \mapsto 1\} \vdash e : \tau, \Psi', L$. Therefore, we have $\Gamma \vdash h$ and $\Psi + \{\rho \mapsto 1\} \vdash R[\rho \mapsto 1]$.

– **App**

By assumptions, we have

- $\Gamma, \Psi \vdash (\lambda x : \tau.e)\ (e') : \tau', \Psi_{\text{keep}} + \Psi_{\text{out}}, L_0 \cup L_1 \cup L$
- $\Gamma \vdash h$
- $\Psi \vdash R$
- $\vdash \Gamma$

By the premise of introduction of $\Gamma, \Psi \vdash (\lambda x : \tau.e)\ (v) : \tau', \Psi_{\text{keep}} + \Psi_{\text{out}}, L$, we have the following conditions.

- $\Gamma, \Psi \vdash \lambda x : \tau.e : (\tau, \Psi_{\text{in}}) \rightarrow^{L} (\tau', \Psi_{\text{out}}), \Psi, \emptyset$
- $\Gamma, \Psi \vdash v : \tau, \Psi, \emptyset$

By Lemma 8 and Lemma 5, we have

$$\Gamma, \Psi \vdash e[v/x], \tau', \Psi_{\text{keep}} + \Psi_{\text{out}}, L$$

– **TyApp**

By assumptions, we have

- $\Gamma, \Psi \vdash (\Lambda\rho.v)[\rho'] : \tau, \Psi, L$
- $\Gamma \vdash h$
- $\Psi \vdash R$
- $\vdash \Gamma$
- $(R, h, (\Lambda\rho.v)[\rho']) \rightarrow (R, h, v[\rho'/\rho])$

By Lemma 1, we have $\Gamma, \Psi \vdash v[\rho'/\rho] : \tau[\rho'/\rho], \Psi, L$. Then, $\Gamma \vdash (R, h, v[\rho'/\rho], \tau[\rho'/\rho], \Psi)$.

– **Fix**

By assumption, we have

- $\Gamma, \Psi \vdash \texttt{fix}\ x : \tau.v : \tau, \Psi, \emptyset$
- $\Gamma \vdash h$
- $\Psi \vdash R$
- $\vdash \Gamma$
- $(R, h, \texttt{fix}\ x : \tau.v \rightarrow (R, h, v[\texttt{fix}\ x : \tau.v/x])$

By the derivation of $\Gamma, \Psi \vdash \texttt{fix}x : \tau.v : \tau, \Psi, \emptyset$, we have $\Gamma, x : \tau, \Psi \vdash v : \tau, \Psi, \emptyset$.
By Lemma 10, we have $\Gamma, \Psi \vdash v[\texttt{fix}\ x : \tau.v : \tau/x], \Psi, \emptyset$.

– **Context**

By structural induction on evaluation contexts.

- E=[]
  Trivial.
- $E = E'\ (e')$
  By assumption, there exist $\Psi$ and $L$ such that
  1. $\Gamma, \Psi \vdash E'[e]\ (e') : \tau_{\text{out}}, \Psi_{\text{keep}} + \Psi_{\text{out}}, L_1 \cup L_2 \cup L$
  2. $\Gamma \vdash h$
  3. $\Psi \vdash R$
  4. $\vdash \Gamma$

By property 1, we have the following derivation.

$$\frac{\Gamma, \Psi \vdash E'[e] : (\Psi_{\text{in}}, \tau_{\text{in}}) \to^{L} (\Psi_{\text{out}}, \tau_{\text{out}}), \Psi_1, L_1 \qquad \Gamma, \Psi_1 \vdash e' : \tau_{\text{in}}, \Psi_{\text{keep}} + \Psi_{\text{in}}, L_2}{\Gamma, \Psi \vdash E'[e]\ (e') : \tau_{\text{out}}, \Psi_{\text{keep}} + \Psi_{\text{out}}, L_1 \cup L_2 \cup L}\ \textbf{App}$$

Then, we have $\Gamma' \vdash (R, h, E'[e], (\Psi_{\text{in}}, \tau_{\text{in}}) \to^{L} (\Psi_{\text{out}}, \tau_{\text{out}}), \Psi_1)$. By I.H., we have $\Gamma' \vdash (R', h', E'[e_n], (\Psi_{\text{in}}, \tau_{\text{in}}) \to^{L} (\Psi_{\text{out}}, \tau_{\text{out}}), \Psi_1)$. We have the following derivation

$$\frac{\Gamma', \Psi' \vdash E'[e_n] : (\Psi_{\text{in}}, \tau_{\text{in}}) \to^{L} (\Psi_{\text{out}}, \tau_{\text{out}}), \Psi_1, L_1 \qquad \Gamma', \Psi_1 \vdash e' : \tau_{\text{in}}, \Psi_{\text{keep}} + \Psi_{\text{in}}, L_2}{\Gamma', \Psi' \vdash E'[e_n]\ (e') : \tau_{\text{out}}, \Psi_{\text{keep}} + \Psi_{\text{out}}, L_1 \cup L_2 \cup L}\ \textbf{App}$$

Hence, we have $\Gamma' \vdash (R', h', E[e_n], \tau_{\text{out}}, \Psi_{\text{keep}} + \Psi_{\text{out}})$.

- $E = v\ (E')$
  By assumption, there exist $\Psi$ and $L$ such that
  1. $\Gamma, \Psi \vdash v\ (E'[e]) : \tau_{\text{out}}, \Psi_{\text{keep}} + \Psi_{\text{out}}, L_1 \cup L_2 \cup L$
  2. $\Gamma \vdash h$
  3. $\Psi \vdash R$
  4. $\vdash \Gamma$

  By property 1, we have following derivation.

$$\frac{\Gamma, \Psi \vdash v : (\Psi_{\text{in}}, \tau_{\text{in}}) \to^{L} (\Psi_{\text{out}}, \tau_{\text{out}}), \Psi, \emptyset \qquad \Gamma, \Psi \vdash E'[e] : \tau_{\text{in}}, \Psi_{\text{keep}} + \Psi_{\text{in}}, L_2}{\Gamma, \Psi \vdash v\ (E'[e]) : \tau_{\text{out}}, \Psi_{\text{keep}} + \Psi_{\text{out}}, L_2 \cup L}\ \textbf{App}$$

  Then, $\Gamma \vdash (R, h, E'[e], \tau_{\text{in}}, \Psi_{\text{keep}} + \Psi_{\text{in}})$. By I.H., we have $\Gamma' \vdash (R', h', E'[e_n], \tau_{\text{in}}, \Psi_{\text{keep}} + \Psi_{\text{in}})$. Hence, we have $\Gamma' \vdash (R', h', E[e_n], \tau_{\text{out}}, \Psi_{\text{keep}} + \Psi_{\text{out}})$.
- $E = E'[\rho]$
- $E = \texttt{ref}\ E'@\rho$
- $E = E'\ \texttt{:=}\ e$
- $E = v\ \texttt{:=}\ E'$
- $E = !E'$

  Proved by the similar approaches.

**Lemma 3.** *Suppose $e$ is a closed term, and we have$\Gamma \vdash (R, h, e, \tau, \Psi)$. Then, either $e$ is a value or $(R, h, e) \to (R', h', e')$.*

*Proof.* By induction on typing derivations.

- **Var**
  We assume $e$ is a closed term, then this case wouldn't occur.
- **Loc**
- **Int**
- **Fun**
- **RegAbs**
  $e$ is a value.

– **Ref**
By assumption, we have the following derivation.

$$\frac{\Gamma, \Psi \vdash e : \tau, \Psi', L \qquad \Psi'(\rho) > 0}{\Gamma, \Psi \vdash \texttt{ref}\ e@\rho : \mathit{ref}(\tau)@\rho, \Psi', L}$$

If $e$ is a value, since $\Psi \vdash R$, $\Psi = \Psi'$ (by Lemma 6), and $\Psi'(\rho) > 0$, that is, $R(\rho) = 1$, for any *ell* such that $\ell \notin \mathit{dom}(h)$, we have $(R, h, \texttt{ref}\ e@\rho) \to (R, h[\ell \mapsto e], \ell@\rho)$.
If $e$ is not a value, by $\Gamma \vdash (R, h, e, \mathit{ref}(\tau)@\rho, \Psi')$ and I.H., we have $(R, h, e) \to (R', h', e')$. Then, we have $(R, h, \texttt{ref}\ e@\rho) \to (R', h', \texttt{ref}\ e'@\rho)$.
– **Write**

$$\frac{\Gamma, \Psi \vdash e : \mathit{ref}(\tau)@\rho, \Psi'', L \qquad \Gamma, \Psi'' \vdash e' : \tau, \Psi', L' \qquad \Psi'(\rho) > 0}{\Gamma, \Psi \vdash e := e' : \mathit{int}, \Psi', L \cup L'}$$

Suppose $e$ and $e'$ are values. By $\Psi \vdash R$, $\Psi = \Psi' = \Psi''$, and $\Psi'(\rho) > 0$, that is, $R(\rho) = 1$. $v = h(\ell)$ by $\Gamma \vdash h$. Otherwise, trivial.
– **Deref**

$$\frac{\Gamma, \Psi \vdash e : \mathit{ref}(\tau)@\rho, \Psi', L \qquad \Psi'(\rho) > 0}{\Gamma, \Psi \vdash !e : \tau, \Psi', L}$$

Suppose that $e$ is a value. Let $\ell@\rho = e$. By $\Psi \vdash R$, $\Psi = \Psi'$, and $\Psi'(\rho) > 0$, $R(\rho) = 1$. $v = h(\ell)$. Otherwise, by $\Gamma \vdash (R, h, e, \mathit{ref}(\tau)@\rho, \Psi')$ and I.H., $(R, h, e) \to (R', h', e')$. Therefore, we have $(R, h, !e) \to (R', h', !e')$.
– **Free**
By assumption, we have $\Psi \vdash R$ and $\Psi(\rho) > 0$, that is, $R(\rho) = 1$. Hence, we have $(R, h, \texttt{free}\ \rho) \to (R[\rho \mapsto 0], h, 0)$.
– **Letreg**

$$\frac{\Gamma, \Psi + \{\rho \mapsto 1\} \vdash e : \tau, \Psi', L \qquad \rho \notin \mathrm{free}(\Gamma) \cup \mathrm{free}(\tau) \qquad \Psi(\rho) = \Psi'(\rho) = 0}{\Gamma, \Psi \vdash \texttt{letreg}\ \rho\ \texttt{in}\ e : \tau, \Psi', L \setminus \{\rho\}}$$

Let $\rho$ be a region variable such that $\rho \notin \mathit{dom}(R)$. We have $(R, h, \texttt{letreg}\rho\texttt{in}e) \to (R[\rho \mapsto 1], h, e)$.
– **App**
– **TyApp**
– **Fix**
Obvious.

**Theorem 1.** *Suppose $\emptyset \vdash (\emptyset, \emptyset, e, \tau, \Psi')$. Then, $e$ is memory safe.*

*Proof.* By Lemma 2, Lemma 3, and Definition 1, it is proven trivially.

**Definition 9.** *We write $\eta \models C$ if*

– *for all $\psi(\rho) = \psi'(\rho) \in C$, $\eta(\psi)(\rho) = \eta(\psi')(\rho)$*

– *for all* $\psi(\rho) > 0 \in C$, $\eta(\psi)(\rho) > 0$
– *for all* $\varphi = \varphi' \in C$, $\eta(\varphi) = \eta(\varphi')$
– *for all* $\psi = \psi' + \{\rho_f \mapsto 1\} \in C$, $\eta(\psi) = \eta(\psi') + \{\rho_f \mapsto 1\}$
– *for all* $\psi(\rho) = 0 \in C$, $\eta(\psi)(\rho) = 0$
– *for all* $\varphi_0 = \varphi_1 + \varphi_2 \in C$, $\eta(\varphi_0) = \eta(\varphi_1) + \eta(\varphi_2)$

**Lemma 11.** *Suppose* $\vdash \sigma =_u \sigma' \Rightarrow C$, $C \subseteq C'$, *and* $\eta \models C'$. *Then,* $\eta(\sigma) = \eta(\sigma')$.

*Proof.* By induction on unification rules and the definition of $\eta \models C'$, it is proven trivially.

**Lemma 12.** *Suppose* $\vdash \sigma =_u \sigma' \Rightarrow C$ *and* $\eta(\sigma) = \eta(\sigma')$. *Then,* $\eta \models C$.

*Proof.* By induction on unification rules and the definition of $\eta \models C$, it is proven trivially.

**Lemma 13.** *Suppose* $\eta \models C$ *and* $\eta \models C'$. *Then,* $\eta \models C \cup C'$.

*Proof.* By the definition of $\eta \models C$, trivial.

**Theorem 2 (Soundness of constraint generation).**
*Suppose* $\Delta, \psi_{in} \vdash e : \sigma, \psi_{out}, L \Rightarrow C$, $C \subseteq C'$ ,*and* $\eta \models C'$. *Then,* $\eta(\Delta), \eta(\psi) \vdash \eta(e) : \eta(\sigma), \eta(\psi_{out}), L$.

*Proof.* By induction on constraint generation rules. We only show ***c*App** case. Other cases are proven by the same approach, or proven trivially.

For ***c*App** case, by I.H., we have the following conditions.

– $\eta(\Delta), \eta(\psi) \vdash \eta(e_1) : \eta((\varphi_{in}, \sigma_{in}) \rightarrow^L (\varphi_{out}, \sigma_{out})), \eta(\psi_1), L_1$ and
– $\eta(\Delta), \eta(\psi_2) \vdash \eta(e_2) : \eta(\sigma'), \eta(\psi_3), L_2$.

By Lemma 11, we have $\eta(\sigma_{in}) = \eta(\sigma')$.
By definition of $\eta \models C$, we have

– $\eta(\psi_1) = \eta(\psi_2)$,
– $\eta(\psi_3) = \eta(\varphi_{in}) + \eta(\psi_{keep})$, and
– $\eta(\psi') = \eta(\psi_{keep}) + \eta(\varphi_{out})$.

Then, we have $\eta(\Delta), \eta(\psi) \vdash \eta(e_1\ (e_2)) : \eta(\sigma_{out}), \eta(\psi'), L_1 \cup L_2 \cup L$.

**Theorem 3 (Completeness of constraint generation).**
*Suppose* $\Delta, \psi_{in} \vdash e : \sigma, \psi_{out}, L \Rightarrow C$ *and* $\eta(\Delta), \eta(\psi) \vdash \eta(e) : \eta(\sigma), \eta(\psi_{out}), L$. *Then,* $\eta \models C$.

*Proof.* By induction on typing derivations. We only show ***A*pp** case. Other cases are proven by the same approach, or proven trivially. For ***A*pp** case, by I.H., we have

– $\eta \models C_1$ and
– $\eta \models C_2$.

By Lemma 13, $\eta \models C_3$.
By assumption, we have

$$\frac{\begin{array}{c}\eta(\Delta), \eta(\psi) \vdash \eta(e_1) : \eta((\varphi_{in}, \sigma_{in}) \rightarrow^{L} (\varphi_{out}, \sigma_{out})), \eta(\psi_1), L_1 \\ \eta(\Delta), \eta(\psi_2) \vdash \eta(e_2) : \eta(\sigma'), \eta(\psi_3), L_2\end{array}}{\eta(\Delta), \eta(\psi) \vdash \eta(e) : \eta(\sigma_{out}), \eta(\psi'), L_1 \cup L_2 \cup L}$$

By contradiction, suppose the following properties don't hold.

- $\eta(\psi_1) = \eta(\psi_2)$
- $\eta(\psi_3) = \eta(\varphi_{in}) + \eta(\psi_{keep})$
- $\eta(\psi') = \eta(\psi_{keep}) + \eta(\varphi_{out})$

Then, we can't deduce $\eta(\Delta), \eta(\psi) \vdash \eta(e) : \eta(\sigma_{out}), \eta(\psi'), L_1 \cup L_2 \cup L$. This leads to a contradiction. Thus, the properties hold.

Consequently, by Lemma 12, we have

$$\eta \models C_1 \cup C_2 \cup C_3 \cup \{\psi_1 = \psi_2, \psi_3 = \varphi_{in} + \psi_{keep}, \psi' = \psi_{keep} + \varphi_{out}\}$$

## B Omitted Figures

$$(R, h, (\lambda x : \tau.e)\ (v)) \to (R, h, e[v/x]) \quad \textbf{App}$$

$$(R, h, (\Lambda\rho.v)\ (\rho')) \to (R, h, v[\rho'/\rho]) \quad \textbf{TyApp}$$

$$(R, h, \mathtt{fix}\ x : \tau.v) \to (R, h, v[(\mathtt{fix}\ x : \tau.v)/x]) \quad \textbf{Fix}$$

$$\frac{R(\rho) = 1 \quad \ell \notin dom(h)}{(R, h, \mathtt{ref}\ v@\rho) \to (R, h[\ell \mapsto v], \ell@\rho)}\ \textbf{Ref}$$

$$\frac{R(\rho) = 1}{(R, h, !\ell@\rho) \to (R, h, h(\ell))}\ \textbf{Read}$$

$$\frac{R(\rho) = 1}{(R, h, \ell@\rho\ \mathtt{:=}\ v) \to (R, h[\ell \mapsto v], 0)}\ \textbf{Write}$$

$$\frac{\rho \notin dom(R)}{(R, h, \mathtt{letreg}\ \rho\ \mathtt{in}\ e) \to (R[\rho \mapsto 1], h, e)}\ \textbf{Letreg}$$

$$\frac{R(\rho) = 1}{(R, h, \mathtt{free}\ \rho) \to (R[\rho \mapsto 0], h, 0)}\ \textbf{Free}$$

$$\frac{e \to e'}{E[e] \to E[e']}\ \textbf{Context}$$

**Fig. 9.** The operational semantics.

$$n[\rho_1/\rho_2] = n$$

$$x[\rho_1/\rho_2] = x$$

$$\lambda x : \tau.e[\rho_1/\rho_2] = \lambda x : \tau[\rho_1/\rho_2].e[\rho_1/\rho_2]$$

$$\Lambda\rho_1'.e[\rho_1/\rho_2] = \Lambda\rho_1'.e[\rho_1/\rho_2] \quad \text{where } \rho_1' \neq \rho_2$$

$$(\texttt{fix}\, x : \tau.v)[\rho_1/\rho_2] = \texttt{fix}\, x : \tau[\rho_1/\rho_2].(v[\rho_1/\rho_2])$$

$$e[\rho_1'][\rho_1/\rho_2] = e[\rho_1/\rho_2][\rho_1'[\rho_1/\rho_2]]$$

$$e\,(e')[\rho_1/\rho_2] = (e[\rho_1/\rho_2])\,(e'[\rho_1/\rho_2])$$

$$\texttt{ref}\, e@\rho_1'[\rho_1/\rho_2] = \texttt{ref}\, e[\rho_1/\rho_2]@\rho_1'[\rho_1/\rho_2]$$

$$(!e)[\rho_1/\rho_2] = !e[\rho_1/\rho_2]$$

$$(e := e')[\rho_1/\rho_2] = (e[\rho_1/\rho_2] := e'[\rho_1/\rho_2])$$

$$(\texttt{letreg}\, \rho_1' \,\texttt{in}\, e)[\rho_1/\rho_2] = \texttt{letreg}\, \rho_1' \,\texttt{in}\, e[\rho_1/\rho_2] \quad \text{where } \rho_1' \neq \rho_2$$

$$(\texttt{free}\, \rho_1')[\rho_1/\rho_2] = \texttt{free}\, \rho_1'[\rho_1/\rho_2]$$

$$(\ell@\rho_1')[\rho_1/\rho_2] = \ell@\rho_1'[\rho_1/\rho_2]$$

**Fig. 10.** $e[\rho_1/\rho_2]$

$$\frac{\psi \text{ fresh}}{\Delta, \psi \vdash x : \Delta(x), \psi, \emptyset \Rightarrow \emptyset}\ \textbf{cInt} \qquad \frac{\psi \text{ fresh}}{\Delta, \psi \vdash n : \mathit{int}, \psi, \emptyset \Rightarrow \emptyset}\ \textbf{cVar}$$

$$\frac{\psi \text{ fresh} \quad \Delta, \psi_{in} \vdash e : \sigma_{out}, \psi_{out}, L \Rightarrow C}{\Delta, \psi \vdash (\lambda x : \sigma_{in}.e) : (\psi_{in}, \sigma_{in}) \xrightarrow{L} (\psi_{out}, \sigma_{out}), \psi, \emptyset \Rightarrow C \cup \{\psi_{in}(\rho) = \psi_{out}(\rho) \mid \rho \in \mathfrak{L} \setminus L\}}\ \textbf{cFun}$$

where $\mathfrak{L}$ denotes all region occurring in a program

$$\frac{\Delta, \psi \vdash e : \forall \rho.\sigma, \psi', L \Rightarrow C}{\Delta, \psi \vdash e[\rho'] : \sigma[\rho'/\rho], \psi', L \Rightarrow C}\ \textbf{cTyApp} \qquad \frac{\Delta, \psi \vdash v : \sigma, \psi, \emptyset \Rightarrow C}{\Delta, \psi \vdash \Lambda \rho.v : \forall \rho.\sigma, \psi, \emptyset \Rightarrow C}\ \textbf{cRegAbs}$$

$$\frac{\Delta, \psi \vdash e : \sigma, \psi', L \Rightarrow C}{\Delta, \psi \vdash \texttt{ref}\ e@\rho : \mathit{ref}(\sigma)@\rho, \psi', L \Rightarrow C \cup \{\psi'(\rho) > 0\}}\ \textbf{cRef}$$

$$\frac{\Delta, \psi_0 \vdash e : \mathit{ref}(\sigma)@\rho, \psi_1, L \Rightarrow C \qquad \Delta, \psi_2 \vdash e' : \sigma, \psi_3, L' \Rightarrow C'}{\Delta, \psi_0 \vdash e\ \texttt{:=}\ e' : \mathit{int}, \psi_3, L \cup L' \Rightarrow C \cup C' \cup \{\psi_1 = \psi_2, \psi_3(\rho) > 0\}}\ \textbf{cWrite}$$

$$\frac{\Delta, \psi \vdash e : \mathit{ref}(\sigma')@\rho, \psi', L \Rightarrow C}{\Delta, \psi \vdash !e : \sigma', \psi', L \Rightarrow C \cup \{\psi'(\rho) > 0\}}\ \textbf{cDeref}$$

$$\frac{\Delta, \psi \vdash e : \sigma, \psi', L \Rightarrow C \qquad \psi_0 \text{ fresh}}{\Delta, \psi_0 \vdash \texttt{letreg}\ \rho\ \texttt{in}\ e : \sigma, \psi', L \setminus \{\rho\} \Rightarrow C \cup \{\psi_0 + \{\rho \mapsto 1\} = \psi, \psi_0(\rho) = 0, \psi'(\rho) = 0\}}\ \textbf{cLetreg}$$

$$\frac{\psi, \psi' \text{ fresh}}{\Delta, \psi \vdash \texttt{free}\ \rho : \mathit{int}, \psi', \{\rho\} \Rightarrow \{\psi = \psi' + \{\rho \mapsto 1\}\}}\ \textbf{cFree}$$

$$\frac{\begin{array}{c}\Delta, \psi \vdash e_1 : (\varphi_{in}, \sigma_{in}) \xrightarrow{L} (\varphi_{out}, \sigma_{out}), \psi_1, L_1 \Rightarrow C_1 \\ \Delta, \psi_2 \vdash e_3 : \sigma', \psi_3, L_2 \Rightarrow C_2 \qquad \vdash \sigma_{in} =_u \sigma' \Rightarrow C_3 \qquad \psi_{keep} \text{ fresh}\end{array}}{\begin{array}{l}\Delta, \psi \vdash e_1\ (e_2) : \sigma_{out}, \psi', L_1 \cup L_2 \cup L \Rightarrow \\ \qquad C_1 \cup C_2 \cup C_3 \cup \{\psi_1 = \psi_2, \psi_3 = \varphi_{in} + \psi_{keep}, \psi' = \psi_{keep} + \varphi_{out}\}\end{array}}\ \textbf{cApp}$$

**Fig. 11.** Constraint generation rules.

$$\frac{}{\vdash \mathit{int} =_u \mathit{int} \Rightarrow \emptyset} \qquad \frac{\vdash \sigma =_u \sigma' \Rightarrow C}{\vdash \mathit{ref}(\sigma)@\rho =_u \mathit{ref}(\sigma')@\rho \Rightarrow C} \qquad \frac{\vdash \sigma =_u \sigma' \Rightarrow C}{\vdash \forall \rho.\sigma =_u \forall \rho.\sigma' \Rightarrow C}$$

$$\frac{\vdash \sigma_{in} = \sigma'_{in} \Rightarrow C \qquad \vdash \sigma_{out} = \sigma'_{out} \Rightarrow C'}{\vdash ((\varphi_{in}, \sigma_{in}) \xrightarrow{L} (\varphi_{out}, \sigma_{out})) =_u ((\varphi'_{in}, \sigma'_{in}) \xrightarrow{L} (\varphi'_{out}, \sigma'_{out})) \Rightarrow C \cup C' \cup \{\varphi_{in} = \varphi'_{in}, \varphi_{out} = \varphi'_{out}\}}$$

**Fig. 12.** Unification rules.