Principals in Programming Languages: Technical Results

Steve Zdancewic Dan Grossman *
Cornell University

June 16, 1999

Abstract

This is the companion technical report for "Principals in Programming Languages" [20]. See that document for a more readable version of these results.

In this paper, we describe two variants of the simply typed λ-calculus extended with a notion of *principal*. The results are languages in which intuitive statements like "the client must call `open` to obtain a file handle" can be phrased and proven formally.

The first language is a two-agent calculus with references and recursive types, while the second language explores the possibility of multiple agents with varying amounts of type information. We use these calculi to give syntactic proofs of some type abstraction results that traditionally require semantic arguments.

1 Introduction

Programmers often have a notion of *principal* in mind when designing the structure of a program. Examples of such principals include modules of a large system, a host and its clients, and, in the extreme, individual functions. Dividing code into such agents is useful for composing programs. Moreover, with the increasing use of extensible systems, such as web browsers, databases [6], and operating systems [7, 3, 2], this notion of principal becomes critical for reasoning about potentially untrusted agents interfacing with host-provided code.

In this paper, we incorporate the idea of principal into variants of the simply-typed λ-calculus. Doing so allows us to formalize statements about agent interaction. For instance, a client must call `open` to obtain a file handle. As a motivating example, we consider the problem of type abstraction in extensible systems.

Consider a host-provided interface for an abstract type of file handles, fh, and operations to create and use them:

`(* File handle implemented as int *)`
`abstype` fh
`open` $: \mathit{string} \to \mathsf{fh}$
`read` $: \mathsf{fh} \to \mathit{int}$

The principals in this scenario are the host implementation of the interface and its clients. Each principal's "view of the world" corresponds to its knowledge regarding fh. In particular, the host knows that $\mathsf{fh} = \mathit{int}$, while clients do not.

The conventional wisdom is that using abstract datatypes in a type-safe language prevents agents from directly accessing host data. Instead, a client may only manipulate such data via a host-provided interface. To formalize this wisdom, it is necessary to prove theorems that say, "agent code can not violate type abstractions provided by the host". For instance, a client should not be able to treat an object of type fh as though it were an integer, even though the host implements it that way.

*This material is based on work supported in part by the AFOSR grant F49620-97-1-0013, ARPA/RADC grant F30602-1-0317, and National Science Foundation Graduate Fellowships. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not reflect the views of these agencies.

How do we prove such properties? One way of phrasing the result is to say that the agent behaves parametrically with respect to the type fh. Using this observation, we can encode the agent program in a language like Girard's System F [5], the polymorphic λ-calculus [16]:

$$\Lambda\mathsf{fh}.\lambda\mathtt{host} : \{\mathtt{open} : \mathit{string} \to \mathsf{fh}, \mathtt{read} : \mathsf{fh} \to \mathit{int}\}.\mathit{agent\_code}$$

Here, the type fh is held abstract by encoding the agent as a polymorphic function. We can then appeal to Reynolds' parametricity results [17] to conclude that the agent respects the host's interface.

Unfortunately, these representation independence results are proven using semantic arguments based on a model of the language (see Mitchell's work [11], for example). We are unaware of any similar results for languages including multiple features of modern languages, such as references, recursive types, objects, threads, and control operators.

Our calculus circumvents this problem by syntactically distinguishing between agents with different type information. We do this by "coloring" host code and client code with different colors and tracking how these colors intermingle during evaluation. By using different semantics for each principal, we force the client to respect the abstract types provided by the host. This separation of principals provides hooks that enable us to prove some type abstraction properties syntactically.

To see why these new mechanisms are useful, consider the evaluation of our agent code when "linked" against a host implementation:

$$\begin{array}{l}(\Lambda\mathsf{fh}.\lambda\mathtt{host} : \tau_h.\mathit{agent\_code}) \quad \mathit{int} \quad \mathit{host\_code} \quad \longmapsto \\ (\lambda\mathtt{host} : \{\mathit{int}/\mathsf{fh}\}\tau_h.\{\mathit{int}/\mathsf{fh}\}\mathit{agent\_code}) \quad \mathit{host\_code} \quad \longmapsto \\ \{\mathit{host\_code}/\mathtt{host}\}\{\mathit{int}/\mathsf{fh}\}\mathit{agent\_code}\end{array}$$

Where $\tau_h = \{\mathtt{open} : \mathit{string} \to \mathsf{fh}, \mathtt{read} : \mathsf{fh} \to \mathit{int}\}$. In this scheme, linking is encoded as application. In one step of the standard operational semantics, the host-type is substituted throughout the agent code. It is impossible to talk about the type fh remaining abstract within the client because fh is replaced by *int*. After a second step, *host_code* is substituted throughout *agent_code* and all distinctions between principals are lost.

The next section describes a two-agent setting including recursive types and state sufficient for proving interesting properties about the file handle example. Detailed proofs of soundness and some abstraction properties are shown. Section 3 introduces a multiagent calculus without mutable references which provides for multiple agents and abstract types. We then revisit the safety properties and language extensions of Section 2. The final sections conclude with related work and other potential uses for principals in programming languages.

2 The Two-agent Language

2.1 Syntax

This section describes a variant of the simply-typed λ-calculus with two principals, an agent and a host. The language maintains a syntactic distinction between host and agent code throughout evaluation. The host exports one abstract type, t, implemented as type τ_h.

Figure 1 gives the syntax for the two-agent calculus. Types, τ, include a base type (b), the host's abstract type (t), function types, reference types, μ-type variables (α), and recursive types. The terms of the language are agent terms (A), agent values ($\hat{A}$), host terms (H), and host values ($\hat{H}$). The metavariable x_a ranges over agent variables which are disjoint from host variables, ranged over by x_h. The metavariable c ranges over values of base type.

Heap labels (ξ^τ), like constants, are not "owned" by either the agent or the host. They are, however, labeled with the type of the value referred to by the label. The level of abstraction of a given reference is decided when the reference is created. This means that the host must decide *a priori* whether a reference to a value of type τ_h should be abstract. A reference of type τ_h $\mathtt{ref}$ can never alias a reference of type t $\mathtt{ref}$.

It is helpful to think of terms generated by A and H as having different colors (indicated by the subscripts a and h respectively) that indicate to which principal each belongs. As observed in the introduction, agent

$$
\begin{array}{rcl}
\tau & ::= & \mathtt{t} \mid \mathtt{b} \mid \tau \to \tau' \mid \tau\ \mathtt{ref} \\
 & & \mid \alpha \mid \mu\alpha.\tau \\[1em]
A & ::= & x_a \mid c \mid \xi^\tau \mid \lambda x_a{:}\tau.\,A \mid A\ A' \mid \\
 & & \mathtt{ref}_\tau A \mid !A \mid A := A' \mid \\
 & & \mathtt{roll}_{\mu\alpha.\tau}\ A \mid \mathtt{unroll}_{\mu\alpha.\tau}\ A \mid \llcorner H \lrcorner_h^\tau \\[1em]
\hat{A} & ::= & c \mid \lambda x_a{:}\tau.\,A \mid \xi^\tau \mid \ \mid \mathtt{roll}_{\mu\alpha.\tau}\ \hat{A} \mid \\
 & & \llcorner \hat{H} \lrcorner_h^{\mathtt{t}} \\[1em]
H & ::= & x_h \mid c \mid \xi^\tau \mid \lambda x_h{:}\tau.\,H \mid H\ H' \mid \\
 & & \mathtt{ref}_\tau H \mid !H \mid H := H' \mid \\
 & & \mathtt{roll}_{\mu\alpha.\tau}\ H \mid \mathtt{unroll}_{\mu\alpha.\tau}\ H \mid \llcorner A \lrcorner_a^\tau \\[1em]
\hat{H} & ::= & c \mid \lambda x_h{:}\tau.\,H \mid \xi^\tau \mid \mathtt{roll}_{\mu\alpha.\tau}\ \hat{H}
\end{array}
$$

Figure 1: Two-agent Syntax

and host terms mix during evaluation. To keep track of this intermingling, agent terms contain *embedded* host terms of the form $\llcorner H \lrcorner_h^\tau$. Intuitively, the brackets delimit a piece of h-colored code, where H is exported to the agent at type τ. Dually, host terms may contain embedded agents.

The type annotations on the embeddings keep track of values of type $\mathtt{t}$ during execution. In particular, a host term of type τ_h may be embedded in an agent term. If the annotation is $\mathtt{t}$, then the agent has no information about the form of the term inside the embedding. Thus, an embedding with annotation $\mathtt{t}$ containing a host value is an agent value.

In order to keep track of the host's extra information about the type $\mathtt{t}$, we introduce an operation Δ_h that converts a type τ with possible occurrences of $\mathtt{t}$ to a "concrete" view seen by the host. This operation essentially substitutes τ_h for $\mathtt{t}$ in τ except under references. The definition of Δ_h is:

$$
\begin{array}{rcl}
\Delta_h(\mathsf{b}) & = & \mathsf{b} \\
\Delta_h(\mathsf{t}) & = & \tau_h \\
\Delta_h(\tau \to \tau') & = & \Delta_h(\tau) \to \Delta_h(\tau') \\
\Delta_h(\mu\alpha.\tau) & = & \mu\alpha.\Delta_h(\tau) \\
\Delta_h(\tau\ \mathtt{ref}) & = & \tau\ \mathtt{ref}
\end{array}
$$

2.2 Notation

Before describing the semantics, we define some convenient notions. Let e range over both agent and host terms, and let $\hat{e}$ range over both agent and host values. The *color* of e is a if e is an A term; otherwise e's color is h. Note that both terms in a syntactically correct application are the same color. Since the host and agent terms share some semantic rules, we use *polychromatic* rules to range over both agent and host terms. The intention is that all terms mentioned in them have the same color –hence the term polychromatic. Such rules can be thought of as short-hand for two analogous rules, one for each color.

We use κ to range over colors and write κ° for the opposite color, so that $a^\circ = h$ and $h^\circ = a$. Where the color of a variable or typing judgment is clear from context or unimportant, we drop the subscript κ to simplify presentation.

We write $\{e'/x_\kappa\}e$ for the capture-avoiding substitution of e' for x_κ in e. Note that substitution crosses embeddings. Terms are equal up to α-conversion, where substituted variables are of the same color.

2.3 Two-agent Dynamic Semantics

We model the heap (memory) as a partial function $M : Labels \to Values \times Agents$ such that if M is defined on ξ^τ, $M(\xi^\tau) = (\hat{e}, \kappa)$ for some value $\hat{e}$ of color κ. The static semantics ensure that type annotations on labels agree with the value in the heap. The heap may contain self-referential data structures.

Figure 2 describes a small-step operational semantics for the two-agent calculus. The transitions are of the form $(M,\ e) \longmapsto (M',\ e')$, indicating that in the configuration where the heap has shape M, the term e steps in one step to e', possibly updating the heap to M'. The first three polychromatic rules establish a call-by-value semantics, while $(\mathbf{P}\boldsymbol{\mu}\mathbf{3})$ makes `unroll` the left inverse of `roll`. The remainder of the polychromatic rules allow evaluation to continue when the immediate subterms are not values.

The host and agent dynamic rules, described next, either propagate embeddings in an appropriate fashion, or mediate access to the heap.

Rules **(AH)** and **(HA)** allow evaluation to proceed within embeddings. Inside embeddings, the rules for the opposite color apply. These "context switches" ensure that terms evaluate in the appropriate context for their color. If an embedded value is exported to the outer principal at type `b`, the outer agent can strip away the embedding and use that value (rules **(AConst)** and **(HConst)**).

Rules **(AHfn)** and **(HAfn)** maintain the distinction between agent and host code. For example, suppose the agent contains a host function that is being exported at type $\tau^1 \to \tau^2$. In this case the agent *does* know that the embedding contains a function, so the agent can apply it to an argument of a suitable type. If instead the function had been exported at type `t`, the agent would not have been able to apply it. The subtlety is that the host type of the function may be more specific than the agent type, such as when $\tau^1 = \mathtt{t}$.

Thus, **(AHfn)** converts an embedded host function to an agent function with argument of type τ. The body of the agent function is an embedding of the host code, except that, as the argument now comes from the agent, every occurrence of the original argument variable, x_h, is replaced by an embedding of the agent's argument variable, $\llcorner x_a \lrcorner_a^\tau$. This embedding is exported to the host at type τ, the type the host originally expected for the function argument. The rule for hosts, **(HAfn)**, is symmetric, except that because the host may use `t` as τ_h, the type of the argument is refined by applying Δ_h.

The rule **(HStrip)** lets the host "open up" an agent value that is really an embedded host value. This allows the host to use a value that has been embedded abstractly in the agent.

The rules $(\mathbf{A}\mu)$ and $(\mathbf{H}\mu)$, propagate the embeddings through `roll` terms when the annotations on the embedding make this permissible. As with the function annotations, the host rule refines the type.

The agent creates new heap labels using rule **(ARef)**. When the agent dereferences a host's heap entry, **(A!)**, the result depends on the last agent to store to that location. If it was the agent, dereferencing returns the stored value. If the host last wrote to the memory, the contents are returned as an embedded host value. The level of abstraction is determined by the type annotation on the heap label. This allows the host to export references to the agent at more abstract types such that dereference preserves the abstraction. Assignment of an agent value to a location, **(AAssn)**, also marks that heap entry with that agent's color. The rule **(ALab)** is similar to **(HStrip)** in that the agent pulls out an agent label from within nested embeddings. Note that the type annotation on the label is forced by the static semantics to be the same as the annotation on the embedding–in effect, the host and agent must agree about the type of references, even if the host knows more.

The host versions of the rules for creating labels and manipulating references are similar.

The crucial point is that any attempt by the agent to treat a value of type `t` as a function, reference, or constant will lead to a stuck configuration (no rule will apply). More generally, we ensure that any configuration in which an abstract value appears in an "active position" is stuck. This fact, along with the stuck configurations of the simply-typed λ-calculus, is enough to prove the safety properties of Section 2.5.

2.4 Two-agent Static Semantics

Figure 3 describes the static semantics for the two-agent calculus. In the type well-formedness rules, Ξ is a set of μ-type variables currently in scope. The judgment $\Xi \vdash \tau$ istype indicates that τ is a well-formed type with free variables in Ξ. Implicit in the heap well-formedness and term typing judgments are antecedents $\emptyset \vdash \tau$ istype for the types mentioned in the rule.

A heap-type, Ψ, is a set of labels describing M. We think of Ψ as a mapping from labels to their types: $\Psi(\xi^\tau) = \tau$. The judgment $\vdash M : \Psi$ says that the heap, M, is well-formed with type Ψ. The rule **(HAlab)**

$\boxed{\text{Polychrome Steps}}$

$$
\begin{array}{lrcll}
(\mathbf{P1}) & (M,\ e_1\ e_2) & \longmapsto & (M',\ e_1'\ e_2) & \text{if } (M,\ e_1) \longmapsto (M',\ e_1') \\
(\mathbf{P2}) & (M,\ \hat{e}\ e) & \longmapsto & (M',\ \hat{e}\ e') & \text{if } (M,\ e) \longmapsto (M',\ e') \\
(\mathbf{P}\beta) & (M,\ (\lambda x{:}\tau.\, e)\ \hat{e}) & \longmapsto & (M,\ \{\hat{e}/x\}e) & \\
(\mathbf{P}\mu\mathbf{1}) & (M,\ \mathtt{roll}_\tau\ e) & \longmapsto & (M',\ \mathtt{roll}_\tau\ e') & \text{if } (M,\ e) \longmapsto (M',\ e') \\
(\mathbf{P}\mu\mathbf{2}) & (M,\ \mathtt{unroll}_\tau\ e) & \longmapsto & (M',\ \mathtt{unroll}_\tau\ e') & \text{if } (M,\ e) \longmapsto (M',\ e') \\
(\mathbf{P}\mu\mathbf{3}) & (M,\ \mathtt{unroll}_{\mu\alpha.\tau}\ \mathtt{roll}_{\mu\alpha.\tau}\ \hat{e}) & \longmapsto & (M,\ \hat{e}) & \\
(\mathbf{P!}) & (M,\ !e) & \longmapsto & (M',\ !e') & \text{if } (M,\ e) \longmapsto (M',\ e') \\
(\mathbf{PRef1}) & (M,\ \mathtt{ref}_\tau e) & \longmapsto & (M',\ \mathtt{ref}_\tau e') & \text{if } (M,\ e) \longmapsto (M',\ e') \\
(\mathbf{PAssn1}) & (M,\ e_1 := e_2) & \longmapsto & (M',\ e_1' := e_2) & \text{if } (M,\ e_1) \longmapsto (M',\ e_1') \\
(\mathbf{PAssn2}) & (M,\ \hat{e} := e_2) & \longmapsto & (M',\ \hat{e} := e_2') & \text{if } (M,\ e_2) \longmapsto (M',\ e_2')
\end{array}
$$

$\boxed{\text{Agent Steps}}$

$$
\begin{array}{lrcll}
(\mathbf{AH}) & (M,\ \lceil H \rceil_h^\tau) & \longmapsto & (M',\ \lceil H' \rceil_h^\tau) & \text{if } (M,\ H) \longmapsto (M',\ H') \\
(\mathbf{AConst}) & (M,\ \lceil c \rceil_h^{\mathsf{b}}) & \longmapsto & (M,\ c) & \\
(\mathbf{AHfn}) & (M,\ \lceil \lambda x_h{:}\tau.\, H \rceil_h^{\tau^1 \to \tau^2}) & \longmapsto & (M,\ \lambda x_a{:}\tau^1.\, \lceil \{\lceil x_a \rceil_a^\tau / x_h\} H \rceil_h^{\tau^2}) & \\
(\mathbf{A}\mu) & (M,\ \lceil \mathtt{roll}_{\mu\alpha.\tau}\ \hat{H} \rceil_h^{\mu\alpha.\tau'}) & \longmapsto & (M,\ \mathtt{roll}_{\mu\alpha.\tau'}\ \lceil \hat{H} \rceil_h^{\{\mu\alpha.\tau'/\alpha\}\tau'}) & \\
(\mathbf{ARef}) & (M,\ \mathtt{ref}_\tau \hat{A}) & \longmapsto & (M[\xi^\tau \mapsto (\hat{A}, a)],\ \xi^\tau) & (\xi^\tau \text{ fresh}) \\
(\mathbf{AAssn}) & (M,\ \xi^\tau := \hat{A}) & \longmapsto & (M[\xi^\tau \mapsto (\hat{A}, a)],\ \hat{A}) & \\
(\mathbf{A!}) & (M,\ !\xi^\tau) & \longmapsto & \begin{cases} (M,\ \hat{A}) & \text{if } M(\xi^\tau) = (\hat{A}, a) \\ (M,\ \lceil \hat{H} \rceil_h^\tau) & \text{if } M(\xi^\tau) = (\hat{H}, h) \end{cases} & \\
(\mathbf{ALab}) & (M,\ \lceil \xi^\tau \rceil_h^{\tau\ \mathtt{ref}}) & \longmapsto & (M,\ \xi^\tau) &
\end{array}
$$

$\boxed{\text{Host Steps}}$

$$
\begin{array}{lrcll}
(\mathbf{HA}) & (M,\ \lceil A \rceil_a^\tau) & \longmapsto & (M',\ \lceil A' \rceil_a^\tau) & \text{if } (M,\ A) \longmapsto (M',\ A') \\
(\mathbf{HConst}) & (M,\ \lceil c \rceil_a^{\mathsf{b}}) & \longmapsto & (M,\ c) & \\
(\mathbf{HAfn}) & (M,\ \lceil \lambda x_a{:}\tau.\, A \rceil_a^{\tau^1 \to \tau^2}) & \longmapsto & (M,\ \lambda x_h{:}\Delta_h(\tau^1).\, \lceil \{\lceil x_h \rceil_h^\tau / x_a\} A \rceil_a^{\tau^2}) & \\
(\mathbf{HStrip}) & (M,\ \lceil \lceil \hat{H} \rceil_h^{\mathsf{t}} \rceil_a^{\tau_h}) & \longmapsto & (M,\ \hat{H}) & \\
(\mathbf{H}\mu) & (M,\ \lceil \mathtt{roll}_{\mu\alpha.\tau}\ \hat{A} \rceil_a^{\mu\alpha.\tau}) & \longmapsto & (M,\ \mathtt{roll}_{\mu\alpha.\Delta_h(\tau)}\ \lceil \hat{A} \rceil_a^{\{\mu\alpha.\tau/\alpha\}\tau}) & \\
(\mathbf{HRef}) & (M,\ \mathtt{ref}_\tau \hat{H}) & \longmapsto & (M[\xi^\tau \mapsto (\hat{H}, h)],\ \xi^\tau) & (\xi^\tau \text{ fresh}) \\
(\mathbf{HAssn}) & (M,\ \xi^\tau := \hat{H}) & \longmapsto & (M[\xi^\tau \mapsto (\hat{H}, a)],\ \hat{H}) & \\
(\mathbf{H!}) & (M,\ !\xi^\tau) & \longmapsto & \begin{cases} (M,\ \hat{H}) & \text{if } M(\xi^\tau) = (\hat{H}, h) \\ (M,\ \lceil \hat{A} \rceil_a^\tau) & \text{if } M(\xi^\tau) = (\hat{A}, a) \end{cases} & \\
(\mathbf{HLab}) & (M,\ \lceil \xi^\tau \rceil_a^{\tau\ \mathtt{ref}}) & \longmapsto & (M,\ \xi^\tau) &
\end{array}
$$

Figure 2: Two-agent Dynamic Semantics

$\boxed{\text{Program}}$

$$(\textbf{Prog}) \quad \frac{\vdash M : \Psi \quad \Psi;\emptyset \vdash e : \tau}{\vdash (M, e) : \tau}$$

$\boxed{\text{Type Well-Formedness}}$

$$(\textbf{Tvar1}) \quad \Xi \vdash \mathsf{t} \text{ istype} \qquad\qquad (\textbf{Tconst}) \quad \Xi \vdash \mathsf{b} \text{ istype}$$

$$(\textbf{Tvar2}) \quad \frac{}{\Xi \vdash \alpha \text{ istype}}(\alpha \in \Xi) \qquad\qquad (\mathbf{T}\mu) \quad \frac{\Xi \cup \{\alpha\} \vdash \tau \text{ istype}}{\Xi \vdash \mu\alpha.\tau \text{ istype}}(\alpha \notin \Xi)$$

$$(\mathbf{T} \rightarrow) \quad \frac{\Xi \vdash \tau \text{ istype} \quad \Xi \vdash \tau' \text{ istype}}{\Xi \vdash \tau \rightarrow \tau' \text{ istype}} \qquad\qquad (\textbf{Tref}) \quad \frac{\Xi \vdash \tau \text{ istype}}{\Xi \vdash \tau\ \mathtt{ref} \text{ istype}}$$

$\boxed{\text{Heap Well-Formedness}}$

$$(\textbf{Hempty}) \quad \vdash \emptyset : \emptyset$$

$$(\textbf{HAlab}) \quad \frac{\vdash M : \Psi \quad \Psi[\xi^\tau : \tau];\emptyset \vdash_a \hat{A} : \tau}{\vdash M[\xi^\tau \mapsto (\hat{A}, a)] : \Psi[\xi^\tau : \tau]} \qquad (\textbf{HHlab}) \quad \frac{\vdash M : \Psi \quad \Psi[\xi^\tau : \tau];\emptyset \vdash_h \hat{H} : \tau' \quad \Delta_h(\tau) = \tau'}{\vdash M[\xi \mapsto (\hat{H}, h)] : \Psi[\xi^\tau : \tau]}$$

$\boxed{\text{Polychrome Rules}}$

$$(\textbf{Pvar}) \quad \Psi;\Gamma \vdash x : \Gamma(x) \qquad\qquad (\textbf{Pconst}) \quad \Psi;\Gamma \vdash c : \mathsf{b}$$

$$(\textbf{Papp}) \quad \frac{\Psi;\Gamma \vdash e : \tau' \rightarrow \tau \quad \Psi;\Gamma \vdash e' : \tau'}{\Psi;\Gamma \vdash e\ e' : \tau} \qquad\qquad (\textbf{Plab}) \quad \frac{\Psi(\xi^\tau) = \tau}{\Psi;\Gamma \vdash \xi^\tau : \tau\ \mathtt{ref}}$$

$$(\textbf{Proll}) \quad \frac{\Psi;\Gamma \vdash e : \{\mu\alpha.\tau/\alpha\}\tau}{\Psi;\Gamma \vdash \mathtt{roll}_{\mu\alpha.\tau}\ e : \mu\alpha.\tau} \qquad\qquad (\textbf{Punroll}) \quad \frac{\Psi;\Gamma \vdash e : \mu\alpha.\tau}{\Psi;\Gamma \vdash \mathtt{unroll}_{\mu\alpha.\tau}\ e : \{\mu\alpha.\tau/\alpha\}\tau}$$

$\boxed{\text{Agent Rules}}$

$$(\textbf{HinA}) \quad \frac{\Psi;\Gamma \vdash_h H : \tau' \quad \Delta_h(\tau) = \tau'}{\Psi;\Gamma \vdash_a \llcorner H \lrcorner_h^\tau : \tau} \qquad\qquad (\textbf{Afn}) \quad \frac{\Psi;\Gamma[x_a : \tau'] \vdash_a A : \tau}{\Psi;\Gamma \vdash_a \lambda x_a{:}\tau'.\, A : \tau' \rightarrow \tau}$$

$$(\textbf{Aref}) \quad \frac{\Psi;\Gamma \vdash_a A : \tau}{\Psi;\Gamma \vdash_a \mathtt{ref}_\tau A : \tau\ \mathtt{ref}} \qquad\qquad (\textbf{Abang}) \quad \frac{\Psi;\Gamma \vdash_a A : \tau\ \mathtt{ref}}{\Psi;\Gamma \vdash\ !A : \tau}$$

$$(\textbf{Aassn}) \quad \frac{\Psi;\Gamma \vdash_a A : \tau\ \mathtt{ref} \quad \Psi;\Gamma \vdash_a A' : \tau}{\Psi;\Gamma \vdash_a A := A' : \tau}$$

$\boxed{\text{Host Rules}}$

$$(\textbf{AinH}) \quad \frac{\Psi;\Gamma \vdash_a A : \tau}{\Psi;\Gamma \vdash_h \llcorner A \lrcorner_a^\tau : \Delta_h(\tau)} \qquad\qquad (\textbf{Hfn}) \quad \frac{\Psi;\Gamma[x_h : \tau'] \vdash_h H : \tau}{\Psi;\Gamma \vdash_h \lambda x_h{:}\tau'.\, H : \tau' \rightarrow \tau}(\Delta_h(\tau') = \tau')$$

$$(\textbf{Href}) \quad \frac{\Psi;\Gamma \vdash_h H : \tau' \quad \Delta_h(\tau) = \tau'}{\Psi;\Gamma \vdash_h \mathtt{ref}_\tau H : \tau\ \mathtt{ref}} \qquad\qquad (\textbf{Hbang}) \quad \frac{\Psi;\Gamma \vdash_h H : \tau\ \mathtt{ref}}{\Psi;\Gamma \vdash\ !H : \Delta_h(\tau)}$$

$$(\textbf{Hassn}) \quad \frac{\Psi;\Gamma \vdash_h H : \tau\ \mathtt{ref} \quad \Psi;\Gamma \vdash_h H' : \tau' \quad \Delta_h(\tau) = \tau'}{\Psi;\Gamma \vdash_h H := H' : \tau'}$$

Figure 3: Two-agent Static Semantics

ensures that agent heap-values have an appropriate typing. Similarly, **(HHlab)** says that a host can store to a heap location a value whose type is a refinement of the annotation on the label. For example, if $\tau_h = \mathtt{b}$ then the agent may store only embedded host constants (i.e. $\llcorner c \lrcorner_h^{\mathtt{t}}$) to the location $\xi^{\mathtt{t}}$, whereas the host may store a regular constant (c) to the same location. Upon dereference, the agent will always obtain a constant embedded at type $\mathtt{t}$ (see rule **(A!)** of the dynamic semantics).

A typing context, Γ, maps variables (of either color) to types. Judgments $\Psi; \Gamma \vdash_\kappa e : \tau$ say that the term e with color κ is well-formed with type τ relative to heap-type Ψ (for labels occurring in e), and context Γ (for free variables occurring in e). The polychromatic rules are standard, as is the introduction rule for agent functions. For host functions, the only difference is that the type annotation for the argument to a function must be concrete. (We could allow abstract type annotations in the host functions, but then application would need an additional condition to check that the argument to the function is compatible. Since the host knows that $\mathtt{t} = \tau_h$, this does not limit expressiveness.)

The rule **(Plab)** says that heap references always have the type ascribed to the label. The rules for creating references, performing assignment, and dereferencing a label allow the host to refine the type of the value being stored or read, while preventing the agent from doing so.

The interesting typing rules are those for embeddings. Rule **(HinA)** says that an embedded host term, H, exported to the agent at type τ (which may contain occurrences of $\mathtt{t}$) has type τ if the host is able to show that the "actual" type of H is $\Delta_h(\tau)$. In other words, the host may hide type information from the agent by replacing some occurrences of τ_h with $\mathtt{t}$ in the exported type. The rule for agents embedded inside of host terms, **(AinH)**, is dual in that the host refines the types provided by the agent.

2.5 Safety Properties

In this section, we explore properties of the two-agent calculus including soundness and some type abstraction theorems. These properties are not intended to be as general or as "realistic" as possible. Rather, they convey the flavor of some statements that are provable using syntactic arguments.

2.5.1 Two-agent Type Soundness

The following lemmas establish type soundness:

Lemma 2.1 (Canonical Forms) *Assuming* $\Psi; \emptyset \vdash \hat{e} : \tau$, *if* $\tau =$

- $\mathtt{b}$, *then* $\hat{e} = c$ *for some* c.
- $\tau' \to \tau''$, *then* $\hat{e} = \lambda x{:}\tau'.\, e'$ *for some* x *and* e'.
- $\mathtt{t}$, *then* $\hat{e} = \llcorner \hat{H} \lrcorner_h^{\mathtt{t}}$ *for some* $\hat{H}$ *of type* τ_h.
- $\tau\ \mathtt{ref}$, *then* $\hat{e} = \xi^\tau$ *for some label* ξ^τ.
- $\mu\alpha.\tau$, *then* $\hat{e} = \mathtt{roll}_{\mu\alpha.\tau}\ e'$ *for some* e' *of type* $\{\mu\alpha.\tau/\alpha\}\tau$.

Proof (sketch): By inspection of the dynamic semantics and the form of values. □

Lemma 2.2 (Substitution) *If* $\Psi; \Gamma[x : \tau'] \vdash e : \tau$ *and* $\Psi; \emptyset \vdash e' : \tau'$ *and* x *and* e' *have the same color then* $\Psi; \Gamma \vdash \{e'/x\}e : \tau$.

Proof: By induction on the typing derivation for $\Psi; \Gamma[x : \tau'] \vdash e : \tau$. Since x and e' have the same color, the substitution is syntactically well formed. We proceed by cases on the last rule of the derivation, omitting those that follow by straightforward induction:

(Pconst) Immediate, since substitution has no effect.

(Pvar) Then $e = y$ and there are two sub-cases:

$x = y$ Then $\tau = \tau'$ and the result follows by assumption.

$x \neq y$ Then $\{e'/x\}y = y$ and by strengthening, since $x \notin FV(y)$, and the assumptions we have $\Psi; \Gamma \vdash y : \tau$.

(Plab) Follows immediately since substitution has no effect.

(Afn) Then $e = \lambda x_a{:}\tau^1.\, A$ and $\tau = \tau^1 \rightarrow \tau^2$ for some x_a, A, τ^1, and τ^2. Furthermore, it must be the case that $\Psi; \Gamma[x : \tau'][x_a : \tau^1] \vdash_a A : \tau^2$. By the side condition, $x \neq x_a$, so $\{e'/x\}\lambda x_a{:}\tau^1.\, A = \lambda x_a{:}\tau^1.\, \{e'/x\}A$. The inductive hypothesis yields $\Psi; \Gamma[x_a : \tau] \vdash_a \{e'/x\}A : \tau^2$, so by **(Afn)** the result follows.

(Hfn) This case is similar to the one above.

$\square$

Lemma 2.3 (Weakening) *(i) If $\Gamma' \supseteq \Gamma$ then $\Psi; \Gamma \vdash_\kappa e : \tau$ implies $\Psi; \Gamma' \vdash_\kappa e : \tau$.*

(ii) If $\Psi' \supseteq \Psi$ then $\Psi; \Gamma \vdash_\kappa e : \tau$ implies $\Psi'; \Gamma \vdash_\kappa e : \tau$.

Proof (sketch): By induction on the first typing derivation. $\square$

Lemma 2.4 (Preservation) *If $\vdash M : \Psi$, $\Psi; \emptyset \vdash e : \tau$ and $(M,\ e) \longmapsto (M',\ e')$ then for some Ψ', $Dom(M') \supseteq Dom(M)$, $\Psi' \supseteq \Psi$, $\vdash M' : \Psi'$, and $\Psi'; \emptyset \vdash e' : \tau$.*

Proof: The proof proceeds by cases on the transition step taken. The agent steps do not differ significantly from the corresponding host-cases, except that the substitution Δ_h is unnecessary since the agent can never refine type information. Since duplicating the proofs is, for the most part, unilluminating, we show as an example the case for **(AHfn)**; the rest are elided. Similarly, those cases that follow by straightforward induction and the Weakening Lemma are omitted.

(Pβ) Then $e = (\lambda x{:}\tau^1.\, e^1)\hat{e}$ for some τ^1, e^1, $\hat{e}$. By assumption, the last rule used in type-checking e must be **(Papp)**, so it follows that $\Psi; [x : \tau^1] \vdash e^1 : \tau$ and $\Psi; \emptyset \vdash \hat{e} : \tau^1$. By Substitution, it follows that $\Psi; \emptyset \vdash \{\hat{e}/x\}e^1 : \tau$, and, since M doesn't change, the result holds.

(Pμ3) Then $e = \mathtt{unroll}_{\mu\alpha.\tau}\ \mathtt{roll}_{\mu\alpha.\tau}\ \hat{e}$ for some $\hat{e}$. The last two rules in the typing derivation must have been **(Proll)** followed by **(Punroll)**, thus we have $\Psi; \emptyset \vdash \hat{e} : \{\mu\alpha.\tau/\alpha\}\tau$. Since $\Psi; \emptyset \vdash e : \{\mu\alpha.\tau/\alpha\}\tau$ and M doesn't change, we're done.

(HA) In this case, $e = \llcorner A \lrcorner_a^{\tau'}$ for some A and its typing derivation ends in **(AinH)**. We conclude $\Psi; \emptyset \vdash_a A : \tau'$ and that $\tau = \Delta_h(\tau')$. Furthermore, we know $(M,\ A) \longmapsto (M',\ A')$, so by the inductive hypothesis: $\vdash M' : \Psi'$, $\Psi' \supseteq \Psi$, $Dom(M) \subseteq Dom(M')$, and $\Psi'; \emptyset \vdash_a A' : \tau'$. Thus the result follows by **(AinH)**.

(HConst) By **(AinH)** and the definition of Δ_h, e has type $\mathtt{b}$. By **(Pconst)**, $e' = c$ has type $\mathtt{b}$. M doesn't change, so $\Psi' = \Psi$.

(HAfn) Then $e = \llcorner \lambda x_a{:}\tau^1.\, A \lrcorner_a^{\tau^1 \rightarrow \tau^2}$. Since e is well-typed there is a derivation of this form:

$$\dfrac{\dfrac{\Psi; [x_a : \tau^1] \vdash_a A : \tau^2}{\Psi; \emptyset \vdash_a \lambda x_a{:}\tau^1.\, A : \tau^1 \rightarrow \tau^2}}{\Psi; \emptyset \vdash_h \llcorner \lambda x_a{:}\tau^1.\, A \lrcorner_a^{\tau^1 \rightarrow \tau^2} : \Delta_h(\tau^1 \rightarrow \tau^2)}$$

Note that from the definition of Δ_h, we have $\Delta_h(\tau^1 \rightarrow \tau^2) = \Delta_h(\tau^1) \rightarrow \Delta_h(\tau^2)$. Also it is easy to show that $\Delta_h(\Delta_h(\tau^1)) = \Delta_h(\tau^1)$. Thus it suffices to find the premise of the following derivation:

$$\dfrac{\dfrac{\Psi; [x_h : \Delta_h(\tau^1)] \vdash_a \{\llcorner x_h \lrcorner_h^{\tau^1}/x_a\}A : \tau^2}{\Psi; [x_h : \Delta_h(\tau^1)] \vdash_h \llcorner \{\llcorner x_h \lrcorner_h^{\tau^1}/x_a\}A \lrcorner_a^{\tau^2} : \Delta_h(\tau^2)}}{\Psi; \emptyset \vdash_h \lambda x_h{:}\Delta_h(\tau^1).\, \llcorner \{\llcorner x_h \lrcorner_h^{\tau^1}/x_a\}A \lrcorner_a^{\tau^2} : \Delta_h(\tau^1) \rightarrow \Delta_h(\tau^2)}$$

Fortunately, we also have this derivation:

$$\frac{\Psi;[x_h : \Delta_h(\tau^1)] \vdash_h x_h : \Delta_h(\tau^1) \quad \Delta_h(\tau^1) = \Delta_h(\tau^1)}{\Psi;[x_h : \Delta_h(\tau^1)] \vdash_a \llcorner x_h \lrcorner_h^{\tau^1} : \tau^1}$$

By weakening the premise of the original derivation and using the substitution lemma, we conclude the premise of the sufficient derivation.

(HStrip) In this case, $e = \llcorner\llcorner\hat{H}\lrcorner_h^{\mathsf{t}}\lrcorner_a^{\tau_h}$, and since, by definition, $\Delta_h(\tau_h) = \tau_h$, there is a derivation of the form:

$$\frac{\dfrac{\Psi;\emptyset \vdash_h \hat{H} : \tau_h \quad \Delta_h(\mathsf{t}) = \tau_h}{\Psi;\emptyset \vdash_a \llcorner\hat{H}\lrcorner_h^{\mathsf{t}} : \tau_h}}{\Psi;\emptyset \vdash_h \llcorner\llcorner\hat{H}\lrcorner_h^{\mathsf{t}}\lrcorner_a^{\tau_h} : \Delta_h(\tau_h) = \tau_h}$$

The premise of the derivation is sufficient to complete this case of the proof.

(Hμ) Then $e = \llcorner\texttt{roll}_{\mu\alpha.\tau'}\ \hat{A}\lrcorner_a^{\mu\alpha.\tau'}$. By definition, $\Delta_h(\mu\alpha.\tau) = \mu\alpha.\Delta_h(\tau)$, so we have the following derivation:

$$\frac{\dfrac{\Psi;\emptyset \vdash_a \hat{A} : \{\mu\alpha.\tau'/\alpha\}\tau'}{\Psi;\emptyset \vdash_a \texttt{roll}_{\mu\alpha.\tau'}\ \hat{A} : \mu\alpha.\tau'}}{\Psi;\emptyset \vdash_h \llcorner\texttt{roll}_{\mu\alpha.\tau'}\ \hat{A}\lrcorner_a^{\mu\alpha.\tau'} : \Delta_h(\mu\alpha.\tau')}$$

Using the premise above, we also have this derivation:

$$\frac{\dfrac{\Psi;\emptyset \vdash_a \hat{A} : \{\mu\alpha.\tau'/\alpha\}\tau'}{\Psi;\emptyset \vdash_h \llcorner\hat{A}\lrcorner_a^{\{\mu\alpha.\tau'/\alpha\}\tau'} : \{\mu\alpha.\Delta_h(\tau')/\alpha\}\Delta_h(\tau')}}{\Psi;\emptyset \vdash_h \texttt{roll}_{\mu\alpha.\Delta_h(\tau')}\ \llcorner\hat{A}\lrcorner_a^{\{\mu\alpha.\tau'/\alpha\}\tau'} : \mu\alpha.\Delta_h(\tau')}$$

which follows from applying the definition of Δ_h like so: $\Delta_h(\{\mu\alpha.\tau'/\alpha\}\tau') = \{\Delta_h(\mu\alpha.\tau')/\alpha\}\Delta_h(\tau') = \{\mu\alpha.\Delta_h(\tau')/\alpha\}\Delta_h(\tau')$. The result holds because, as already observed, $\Delta_h(\mu\alpha.\tau') = \mu\alpha.\Delta_h(\tau')$.

(HRef) Then $e = \texttt{ref}_{\tau'}\hat{H}$ and $\tau = \tau'\ \texttt{ref}$. Since e is well typed, there is a derivation of the form:

$$\frac{\Psi;\emptyset \vdash_h \hat{H} : \tau'' \quad \Delta_h(\tau') = \tau''}{\Psi;\emptyset \vdash_h \texttt{ref}_{\tau'}\hat{H} : \tau'\ \texttt{ref}}$$

We must show that there exists a $\Psi' \supseteq \Psi$ such that $\vdash M[\xi^{\tau'} \mapsto (\hat{H}, h)] : \Psi'$ and such that $\Psi';\emptyset \vdash_h \xi^{\tau'} : \tau'\ \texttt{ref}$. Consider $\Psi' = \Psi[\xi^{\tau'} : \tau']$. Then by weakening the premise of the above derivation and applying the rule **(HHlab)** we have that $\vdash M[\xi^{\tau'} \mapsto (\hat{H}, h)] : \Psi[\xi^{\tau'} : \tau']$. The rule **(Plab)** allows us to derive the rest of the result.

(Hassn) In this case, $e = \xi^{\tau} := \hat{H}$. Sine it is well typed there is the following derivation:

$$\frac{\dfrac{\Psi(\xi^{\tau}) = \tau}{\Psi;\emptyset \vdash_h \xi^{\tau} : \tau\ \texttt{ref}} \quad \Psi;\emptyset \vdash_h \hat{H} : \tau' \quad \Delta_h(\tau) = \tau'}{\Psi;\emptyset \vdash_h \xi^{\tau} := \hat{H} : \tau}$$

Note that since $\Psi(\xi^{\tau}) = \tau$ we have $\Psi[\xi^{\tau} : \tau] = \Psi$ and, by applying **(HHlab)** to the premises of the above derivation, we have:

$$\frac{\vdash M : \Psi \quad \Psi[\xi^{\tau} : \tau];\emptyset \vdash_h \hat{H} : \tau' \quad \Delta_h(\tau) = \tau'}{\vdash M[\xi^{\tau} \mapsto (\hat{H}, h)] : \Psi[\xi^{\tau} : \tau]}$$

Which shows that the new heap is well typed under the same Ψ. The premise $\Psi;\emptyset \vdash_h \xi^{\tau} : \tau\ \texttt{ref}$ is what we need to complete this case of the proof.

(H!) We have $e = !\xi^{\tau'}$ and since e is well typed, there is this derivation:

$$\dfrac{\dfrac{\Psi(\xi^{\tau'}) = \tau'}{\Psi; \emptyset \vdash_h \xi^{\tau'} : \tau' \ \texttt{ref}}}{\Psi; \emptyset \vdash_h !\xi^{\tau'} : \Delta_h(\tau')}$$

It follows that $\tau = \Delta_h(\tau')$. There are two cases depending on the heap contents at label $\xi^{\tau'}$:

$M(\xi^{\tau'}) = (\hat{H}, h)$ The well-typedness of the heap must have been arrived at by an application of rule **(HHlab)**, which means we have the following derivation:

$$\dfrac{\vdash M : \Psi \quad \Psi[\xi^{\tau'} : \tau']; \emptyset \vdash_h \hat{H} : \tau'' \quad \Delta_h(\tau') = \tau''}{\vdash M[\xi^{\tau'} \mapsto (\hat{H}, h)] : \Psi[\xi^{\tau'} : \tau']}$$

Since $\Psi(\xi^{\tau'}) = \tau'$ we have $\Psi = \Psi[\xi^{\tau'} : \tau']$ and it follows that $\Psi; \emptyset \vdash_h \hat{H} : \Delta(\tau')$, which is exactly what we want.

$M(\xi^{\tau'}) = (\hat{A}, a)$ In this case, the fact that M is well-typed follows from an application of **(HAlab)** with this derivation:

$$\dfrac{\vdash M : \Psi \quad \Psi[\xi^{\tau'} : \tau']; \emptyset \vdash_a \hat{A} : \tau'}{\vdash M[\xi^{\tau'} \mapsto (\hat{A}, a)] : \Psi[\xi^{\tau'} : \tau']}$$

As before, we also know that $\Psi = \Psi[\xi^{\tau'} : \tau']$, so an application of **(AinH)** to the premise above yields $\Psi; \emptyset \vdash_h \llcorner \hat{A} \lrcorner_a^{\tau'} : \Delta_h(\tau')$ which is the desired result.

(Hlab) Here, $e = \llcorner \xi^{\tau'} \lrcorner_a^{\tau' \ \texttt{ref}}$ and we have the derivation:

$$\dfrac{\dfrac{\Psi(\xi^{\tau'}) = \tau'}{\Psi; \emptyset \vdash_a \xi^{\tau'} : \tau' \ \texttt{ref}}}{\Psi; \emptyset \vdash_h \llcorner \xi^{\tau'} \lrcorner_a^{\tau' \ \texttt{ref}} : \Delta_h(\tau' \ \texttt{ref})}$$

But, the definition of Δ_h ensures that $\Delta_h(\tau' \ \texttt{ref}) = \tau' \ \texttt{ref}$. Thus we may use **(Plab)** with the premise of the above derivation to conclude that $\Psi; \emptyset \vdash_h \xi^{\tau'} : \tau' \ \texttt{ref}$. Since M doesn't change, we're done.

(AHfn) In this case, $e = \llcorner \lambda x_h{:}\tau^0.\, H \lrcorner_h^{\tau^1 \to \tau^2}$ and the typing derivation is of the form:

$$\dfrac{\dfrac{\Psi; [x_h : \tau^0] \vdash_h H : \tau^3}{\Psi; \emptyset \vdash_h \lambda x_h{:}\tau^0.\, H : \tau^0 \to \tau^3} \quad \Delta_h(\tau^1 \to \tau^2) = \tau^0 \to \tau^3}{\Psi; \emptyset \vdash_a \llcorner \lambda x_h{:}\tau^0.\, H \lrcorner_h^{\tau^1 \to \tau^2} : \tau^1 \to \tau^2}$$

It suffices to find a derivation of this form for e':

$$\dfrac{\dfrac{\Psi; [x_a : \tau^1] \vdash_h \{\llcorner x_a \lrcorner_a^{\tau^0} / x_h\} H : \tau^3 \quad \tau^3 = \Delta_h(\tau^2)}{\Psi; [x_a : \tau^1] \vdash_a \llcorner \{\llcorner x_a \lrcorner_a^{\tau^0} / x_h\} H \lrcorner_h^{\tau^2} : \tau^2}}{\Psi; \emptyset \vdash_a \lambda x_a{:}\tau^1.\, \llcorner \{\llcorner x_a \lrcorner_a^{\tau^0} / x_h\} H \lrcorner_h^{\tau^2} : \tau^1 \to \tau^2}$$

From the first derivation, it follows that $\tau^3 = \Delta_h(\tau^2)$. Since $\tau^0 = \Delta_h(\tau^0)$, we also have the following derivation:

$$\dfrac{\Psi; [x_a : \tau^0] \vdash_a x_a : \tau^0}{\Psi; [x_a : \tau^0] \vdash_h \llcorner x_a \lrcorner_a^{\tau^0} : \Delta_h(\tau^0) = \tau^0}$$

By weakening the first premise of the derivation and applying the substitution lemma, we conclude the first premise of the second derivation, so the result holds.

□

Lemma 2.5 (Progress) *If $\vdash M : \Psi$ and $\Psi; \emptyset \vdash e : \tau$, then either e is a value or there exists an e' such that $(M,\ e) \longmapsto (M',\ e')$.*

Proof: By structural induction on e. Note that because e is well-typed, e cannot be a variable. We proceed by cases:

$e = c$**,** $e = \lambda x{:}\tau.\, e'$**,** $e = \xi^\tau$**,** $e = \texttt{roll}_{\mu\alpha.\tau}\ \hat{e}$**, or** $e = \llbracket \hat{H} \rrbracket_h^{\mathsf{t}}$ Then e is a value and the lemma holds.

$e = e'e''$ This case follows by straightforward induction in the case that e' or e'' is not a value—the transition is via rule **(P1)** or **(P2)**. In the case that e' and e'' are both values, well-typedness of e and Canonical Forms yields that $e' = \lambda x{:}\tau.\, e'''$ and the transition is via rule **(Pβ)**.

$e = \texttt{ref}_\tau e'$ If e' is not a value, then the transition is via rule **(PRef1)**, otherwise either **(Aref)** or **(Href)** applies, depending on the color of e.

$e = !e'$ If e' is not a value, then rule **(P!)** applies. Otherwise, well typedness of e yields that e' has type τ `ref` for some type τ. Since e' is a value, Canonical Forms tells us that $e' = \xi^\tau$. Either **(A!)** or **(H!)** applies depending on the color of e'.

$e = e' := e''$ If at least one of e' and e'' is not a value, then rules **(PAssn1)** or **(PAssn2)** apply. If both are values, then well-typedness of e yields that e' has type τ `ref`, and so, by Canonical Forms, $e' = \xi^\tau$. One of **(AAssn)** or **(HAssn)** applies, depending on the color of e'.

$e = \texttt{roll}_{\mu\alpha.\tau}\ e'$ If e' is not a value, then **(Pμ1)** applies.

$e = \texttt{unroll}_{\mu\alpha.\tau}\ e'$ If e' is not a value, then **(Pμ2)** applies. Otherwise, well-typedness of e and Canonical Forms yields that $e' = \texttt{roll}_{\mu\alpha.\tau}\ \hat{e}$, in which case **(P$\mu$3)** applies.

$e = \llbracket e' \rrbracket_\kappa^\tau$ If e' is not a value, then, by the inductive hypothesis, either **(AH)** or **(HA)** applies. We consider the remaining cases, assuming that e' is a value but e is not:

- $\tau = \tau'$ `ref` Here, e' must be of the form $\xi^{\tau'}$ (by Canonical Forms) and either **(ALab)** or **(HLab)** applies.
- $\tau = \mathsf{b}$ Then $e = \llbracket c \rrbracket_\kappa^{\mathsf{b}}$ and one of **(AConst)** or **(HConst)** applies, depending on e's color.
- $\tau = \mu\alpha.\tau$ Then $e = \llbracket \texttt{roll}_{\mu\alpha.\tau 2}\ \hat{e} \rrbracket_\kappa^{\mu\alpha.\tau}$ by Canonical Forms on e' and it follows that either **(Aμ)** or **(Hμ)** applies.
- $\tau = \tau^1 \to \tau^2$ By Canonical forms, $e' = \lambda x{:}\tau'.\, e''$ and either **(AHfn)** or **(HAfn)** provides the transition.
- $\kappa = a, e' = \llbracket \hat{H} \rrbracket_h^{\mathsf{t}}$ In this case, **(HStrip)** applies.

□

Definition 2.6 (Stuck Terms) *A configuration $(M,\ e)$ is <u>stuck</u> if e is not a value and there is no configuration $(M',\ e')$ such that $(M,\ e) \longmapsto (M',\ e')$.*

Theorem 2.7 (Type Soundness) *If $\Psi; \emptyset \vdash e : \tau$ and $\vdash M : \Psi$ then there is no stuck $(M',\ e')$ such that $(M,\ e) \longmapsto^* (M',\ e')$.*

Proof: Follows from Preservation and Progress. □

$$
\begin{array}{rcl}
erase(x) & = & x \\
erase(c) & = & c \\
erase(\xi^\tau) & = & \xi \\
erase(\lambda x{:}\tau.\, e) & = & \lambda x{:}\{\tau_h/\mathtt{t}\}\tau.\, erase(e) \\
erase(e\ e') & = & erase(e)\ erase(e') \\
erase(\mathtt{ref}_\tau e) & = & \mathtt{ref}_{\{\tau_h/\mathtt{t}\}\tau} erase(e) \\
erase(e := e') & = & erase(e) := erase(e') \\
erase(!e) & = & !erase(e) \\
erase(\mathtt{roll}_{\mu\alpha.\tau}\ e) & = & \mathtt{roll}_{\{\tau_h/\mathtt{t}\}\mu\alpha.\tau}\ erase(e) \\
erase(\mathtt{unroll}_{\mu\alpha.\tau}\ e) & = & \mathtt{unroll}_{\{\tau_h/\mathtt{t}\}\mu\alpha.\tau}\ erase(e) \\
erase(\llcorner e \lrcorner^\tau_\kappa) & = & erase(e) \\
 & & \\
erase(\emptyset) & = & \emptyset \\
erase(M[\xi^\tau \mapsto (\hat{e}, \kappa)]) & = & erase(M)[\xi \mapsto erase(\hat{e})] \\
 & & \\
erase(M,\ e) & = & (erase(M),\ erase(e))
\end{array}
$$

Figure 4: Two-agent *erase* translation

2.5.2 Two-agent Erasure

Given a term, if we ignore the distinction between colors, erase the embeddings, and replace $\mathtt{t}$ with τ_h, we have a simply-typed λ-calculus term. Formally, Figure 4 defines the erasure of a two-agent term. (All rules are polychromatic.) The following lemma states that erasure commutes with evaluation.

Lemma 2.8 (Erasure) *Let e be any two-agent term such that $\Psi; \emptyset \vdash e : \tau$. Then $(M,\ e) \longmapsto^* (M',\ \hat{e})$ iff $erase(M,\ e) \longmapsto^* erase(M',\ \hat{e})$.*

Proof (sketch): First prove that substitution commutes with erasure. With that lemma in hand, the erasure lemma follows by induction on the source derivation. The proof is straightforward and not interesting so we omit it here. $\square$

2.5.3 Two-agent Abstraction Properties

Before proving the type abstraction properties, we make a convenient definition, which allows us to perform substitutions in an open heap:

Definition 2.9 (Pre-Heap) *If Γ is a typing context, we say that a finite map $P : Labels \to (Terms \times Agents)$ is a Γ-pre-heap of type Ψ, written $\Gamma \vdash P : \Psi$, if for every label ξ^τ such that $P(\xi^\tau) = (A, a)$ it is the case that $\Psi; \Gamma \vdash_a A : \tau$ and for every label ξ^τ such that $P(\xi^\tau) = (H, h)$ it is the case that $\Psi; \Gamma \vdash_h H : \tau'$ and $\Delta_h(\tau) = \tau'$.*

Note that if $\Gamma = \emptyset$ then a pre-heap is just a normal heap. The difference is that pre-heaps can contain references to open terms in addition to values. Substitution over a Γ-pre-heap P is as follows: If $\Psi; \emptyset \vdash_\kappa \hat{e} : \tau$ and $\Gamma \vdash P : \Psi$ and $\Gamma(x_\kappa) = \tau$ then $\{\hat{e}/x_\kappa\}P$ is defined by:

- $\{\hat{e}/x_\kappa\}\emptyset = \emptyset$
- $\{\hat{e}/x_\kappa\}(P[\xi^{\tau'} \mapsto (e, \kappa')]) = (\{\hat{e}/x_\kappa\}P)[\xi^{\tau'} \mapsto (\{\hat{e}/x_\kappa\}e, \kappa')]$

Lemma 2.10 (Heap Substitution) *If $\Gamma[x_\kappa : \tau] \vdash P : \Psi$ and $\Gamma(x_\kappa) = \tau$ and $\Psi; \emptyset \vdash_\kappa \hat{e} : \tau$ then $\Gamma \vdash \{\hat{e}/x_\kappa\}P : \Psi$.*

We write $\{e/x_\kappa\}(P,\ e')$ for $(\{e/x_\kappa\}P,\ \{e/x_\kappa\}e')$ whenever such a substitution is appropriate.

Definition 2.11 (Reachable Term) *A term e' is* reachable *from a configuration $(M,\ e)$ if either*

- *e' is a subterm of e*
- *For any label ξ^τ, a subterm of e, $M(\xi^\tau) = (\hat{e}, \kappa)$ and e' is reachable from $(M,\ \hat{e})$.*

Definition 2.12 (Host-free) *If no host-term is reachable from the configuration $(M,\ A)$ then $(M,\ A)$ is said to be* host-free*.*

We use the terminology reachable and host-free (defined as above) also for pairs $(P,\ A)$ where P is a pre-heap and A is a (not-necessarily-closed) expression.

Lemma 2.13 (Value Abstraction) *Let P be a pre-heap such that $[x_a : \mathtt{t}] \vdash P : \Psi$. Let $\hat{H}$ and $\hat{H}'$ be host values such that $\Psi; \emptyset \vdash_h \hat{H} : \tau_h$ and $\Psi; \emptyset \vdash_h \hat{H}' : \tau_h$. Let A be an agent-term such that $\Psi; [x_a : \mathtt{t}] \vdash_a A : \tau$ and further suppose that $(P,\ A)$ is host-free and $\{\llcorner \hat{H} \lrcorner_h^{\mathtt{t}} / x_a\}A$ is not a value. Let $\sigma_1 = \{\llcorner \hat{H} \lrcorner_h^{\mathtt{t}} / x_a\}$ and $\sigma_2 = \{\llcorner \hat{H}' \lrcorner_h^{\mathtt{t}} / x_a\}$. Then there exists a term A' and a $[x_a : \mathtt{t}]$-pre-heap P' such that:*

1. *$[x_a : \mathtt{t}] \vdash P' : \Psi'$ where $\Psi' \supseteq \Psi$*
2. *$\Psi'; [x_a : \mathtt{t}] \vdash_a A' : \tau$*
3. *$(P',\ A')$ is host-free*
4. *$\sigma_1(P,\ A) \longmapsto \sigma_1(P',\ A')$*
5. *$\sigma_2(P,\ A) \longmapsto \sigma_2(P',\ A')$*

Proof: The proof goes by induction on the derivation that $\sigma_1(p,\ A)$ transitions. Note that since A is host-free and $\hat{H}$ is a value, the only rules occurring in the derivation are polychromatic or agent-rules. We can eliminate all of the agent rules except **(ARef)**, **(AAssn)** and **(A!)** since $\hat{H}$ is a value and there are no other reachable host-terms. Thus we proceed by cases on each of the remaining rules:

(P1), (P2), (Pμ1), (Pμ2), (P!), (PRef1), (PAssn1), or (PAssn2) These follow by straightforward induction and the definition of substitution. As an example, we show the case for **(P1)**; the rest are similar. Since $\sigma_1 A$ transitions via rule **(P1)** and since $\llcorner \hat{H} \lrcorner_h^{\mathtt{t}}$ is a value, A must be of the form $A^1\ A^2$ where $\sigma_1 A^1$ is not a value. This follows because $\sigma_1 A = \sigma_1(A^1\ A^2) = (\sigma_1 A^1)\ (\sigma_1 A^2)$ by the definition of substitution. Furthermore, the typing rule for applications yields that $\Psi; [x_a : \mathtt{t}] \vdash_a A^1 : \tau' \to \tau$ for some τ'. Since A^1 is a subterm of A, it follows from the definition of host-free that $(P,\ A^1)$ is host-free. Thus the inductive hypothesis applies and we conclude that there is a term A'' and a $[x_a : \mathtt{t}]$-pre-heap P'' such that $\Psi''; [x_a : \mathtt{t}] \vdash_a A'' : \tau' \to \tau$, $(P'',\ A'')$ is host-free, $\sigma_1(P,\ A^1) \longmapsto \sigma_1(P'',\ A'')$, and $\sigma_2(P,\ A^1) \longmapsto \sigma_2(P'',\ A'')$. Let $\Psi' = \Psi''$ and choose $P' = P''$ and $A' = (A''\ A^2)$. Claim: P' and A' satisfy properties 1 through 5. Property 1 holds by the inductive hypothesis and weakening since $\Psi' = \Psi'' \supseteq \Psi$ and $P' = P''$. Property 2 follows from the well-typedness of A and the inductive hypothesis via rule **(Papp)**. Property 3 follows from the inductive hypothesis and the observation that since A^2 is a subterm of A it is host-free. Property 4 is established by an application of **(P1)** to the results of the inductive hypothesis: $(\sigma_1 P,\ (\sigma_1 A^1)\ (\sigma_1 A^2)) \longmapsto (\sigma_1 P,\ (\sigma_1 A'')\ (\sigma_1 A^2))$ which equals $(\sigma_1 P,\ \sigma_1(A''\ A^2))$ which is $\sigma_1(P',\ A')$. Property 5 follows analogously.

(Pβ) Since it is well-typed, and, in particular, because x_a has type $\mathtt{t}$ and so can't appear in the application position of an agent term, $\sigma_1 A$ is of the form $\sigma_1((\lambda y_a{:}\tau.\, A^1)\ A^2)$ which, if $x_a \neq y_a$, equals (by the definition of substitution) $(\lambda y_a{:}\tau'.\, \sigma_1 A^1)\ (\sigma_1 A^2)$, or, if $x_a = y_a$, it equals $(\lambda y_a{:}\tau'.\, A^1)\ (\sigma_1 A^2)$. In the former case, the transition step yields $\{(\sigma_1 A^2)/y_a\}(\sigma_1 A^1)$, which, since $x_a \neq y_a$, is $\sigma_1(\{A^2/y_a\}A^1)$, while in the latter case the transition yields $\{(\sigma_1 A^2)/y_a\}A^1 = \sigma_1(\{A^2/y_a\}A^1$. Choose $A' = \{A^2/y_a\}A^1$ and $P' = P$. Again, since P is unchanged, Property 1 holds by assumption. Property 2: This follows from the Substitution Lemma and the assumption that A is well-typed. Property 3: P' is clearly host-free since P is. A' has no occurrences of host code since neither A^1 nor A^2 do and substitution doesn't change the color of code. Property 4: follows from observations above about the transition step. Property 5: This follows analogously to 4 using the observation that $\sigma_2 A = \sigma_2(\lambda y_a{:}\tau'.\, A^1)\ (\sigma_2 A^2)$ and following similar calculations of the transition step.

(Pμ3) In this case, $\sigma_1 A$ is of the form $\sigma_1(\texttt{unroll}_{\mu\alpha.\tau}\ \texttt{roll}_{\mu\alpha.\tau'}\ A^1)$. Take $A' = A^1$ and $P' = P$. Property 1 is trivial. Property 2 follows from the well-typedness of A, since the last two steps of the typing derivation must have been **(Proll)** followed by **(Punroll)**. We conclude that $\Psi; [x_a : \mathsf{t}] \vdash_a A' : \{\mu\alpha.\tau'/\alpha\}\tau'$. Since A' is a sub-term of A, the configuration $(P',\ A')$ is host-free, satisfying Property 3. Property 4 is established via the definition of substitution: $(\sigma_1 P,\ \sigma_1 A) = (\sigma_1 P,\ \texttt{unroll}_{\mu\alpha.\tau'}\ \texttt{roll}_{\mu\alpha.\tau'}\ (\sigma_1 A^1))$ which steps via **(Pμ3)** to $(\sigma_1 P,\ \sigma_1 A^1) = \sigma_1(P,\ A^1)$ which is just $\sigma_1(P',\ A')$. Similarly we have $(\sigma_2 P,\ \sigma_2 A) = (\sigma_2 P,\ \texttt{unroll}_{\mu\alpha.\tau'}\ \texttt{roll}_{\mu\alpha.\tau'}\ (\sigma_2 A^1))$ which steps via **(Pμ3)** to $(\sigma_2 P,\ \sigma_2 A^1) = \sigma_2(P,\ A^1)$, again, just $\sigma_2(P',\ A')$.

(ARef) In this case, $\sigma_1 A = \texttt{ref}_\tau \hat{A}$ and it follows that $A = \texttt{ref}_\tau A''$ for some A''. Since $\sigma_1(P,\ A)$ steps to $(M,\ \xi^\tau)$ for some M and fresh label, ξ^τ, we choose $P' = P[\xi \mapsto (A'', a)]$ and $A' = \xi^\tau$. It remains to verify properties 1 through 5. Property 1: By assumption we have $\Psi; [x_a : \mathsf{t}] \vdash_a \texttt{ref}_\tau A'' : \tau\ \texttt{ref}$, which must have been derived by an application of **(Aref)**. It follows that $\Psi; [x_a : \mathsf{t}] \vdash_a A'' : \tau$. Since P is an $[x_a : \mathsf{t}]$-pre-heap with type Ψ and $\xi^\tau \notin Dom(P)$, it follows that $[x_a : \mathsf{t}] \vdash P[\xi \mapsto A''] : \Psi' = \Psi[\xi^\tau : \tau]$, and hence $\Psi' \supseteq \Psi$. Property 2: We must show that $\Psi'; [x_a : \mathsf{t}] \vdash_a \xi^\tau : \tau\ \texttt{ref}$. We may conclude this via rule **(Plab)** since $\Psi'(\xi^\tau) = \tau$ as constructed above. Property 3: Since P and A'' are host-free and the only terms reachable from $(P',\ \xi^\tau)$ are those reachable from $(P',\ A'')$ plus ξ^τ (an agent label), it follows that $(P',\ A')$ is host-free as well. We verify Property 4 with the following calculation: $\sigma_1(P,\ A) = (\sigma_1 P,\ \sigma_1 A) = (\sigma_1 P,\ \sigma_1(\texttt{ref}_\tau A'')) = (\sigma_1 P,\ \texttt{ref}_\tau \sigma_1 A'') \longmapsto ((\sigma_1 P)[\xi^\tau \mapsto (\sigma_1 A'', a)],\ \xi^\tau) = (\sigma_1(P[\xi^\tau \mapsto (A'', a)]),\ \xi^\tau) = \sigma_1(P[\xi^\tau \mapsto (A'', a)],\ \xi^\tau) = \sigma_1(P',\ A')$. Property 5 follows via the same calculation replacing $\sigma_1$by σ_2.

(AAssn) In this case, $\sigma_1 A = \xi^\tau := \hat{A}$ it follows that $A = \xi^\tau := A''$ where A'' is host-free. Take $A' = A''$ and choose $P' = P[\xi^\tau \mapsto (A', a)]$. Note that $\Psi; [x_a : \mathsf{t}] \vdash_a \xi^\tau := A'' : \tau$ must have been derived via rule **(AAssn)** and so it follows that $\Psi; [x_a : \mathsf{t}] \vdash_a \xi^\tau : \tau\ \texttt{ref}$ which can be derived only via **(Alab)**. Thus it must be the case that $\Psi(\xi^\tau) = \tau$. Likewise it must be the case that $\Psi; [x_a : \mathsf{t}] \vdash_a A'' : \tau$. We first verify Property 1: By assumption, $[x_a : \mathsf{t}] \vdash P : \Psi$. By the definition of pre-heap and the type derivation for $A'' = A'$, it follows that $[x_a : \mathsf{t}] \vdash P[\xi^\tau \mapsto (A', a)] : \Psi$. Property 2 has already been observed to be a consequence of the original typing derivation. Property 3 follows since A' is host free and the original P was also host-free. We verify Property 4 by the calculation: $\sigma_1(P,\ A) = \sigma_1(P,\ \xi^\tau := A'') = (\sigma_1 P,\ \sigma_1(\xi^\tau := A'')) = (\sigma_1 P,\ \xi^\tau := \sigma_1 A'') \longmapsto ((\sigma_1 P)[\xi^\tau \mapsto (\sigma_1 A'', a)],\ \sigma_1 A'') = (\sigma_1(P[\xi^\tau \mapsto (A'', a)]),\ \sigma_1 A'') = \sigma_1(P[\xi^\tau \mapsto (A'', a)],\ A'') = \sigma_1(P',\ A')$. Property 5 follows analogously.

(A!) In this case, $\sigma_1 A = !\xi^\tau$, which means that $A = !\xi^\tau$. Since $(P,\ A)$ is host-free, it must be the case that $P(\xi^\tau) = (A', a)$ for some A'. This is the A' we seek, it also means that the transition step is determined to be of the first flavor listed in **(A!)**. Let $P' = P$. Then Property 1 follows since P is unchanged. Property 2 follows by the well-typedness of the pre-heap P. Property 3 holds since ξ^τ is a subterm of A which means that A' is reachable from A and hence $(P,\ A')$ is also host-free. Property 4 is established as follows: $\sigma_1(P,\ A) = (\sigma_1 P,\ \sigma_1 A) = (\sigma_1 P,\ !\xi^\tau)$. As noted above, $P(\xi^\tau) = (A', a)$ implies that this must step via **(A!)** to $(\sigma_1 P,\ (\sigma_1 A')) = \sigma_1(P',\ A')$. Property 4 follows by a similar computation.

□

Using the Value Abstraction lemma, we prove the following:

Theorem 2.14 (Independence of Evaluation) *Let M be a heap such that $\vdash M : \Psi$. Let $\hat{H}$ and $\hat{H}'$ be host values such that $\Psi; \emptyset \vdash_h \hat{H} : \tau_h$ and $\Psi; \emptyset \vdash_h \hat{H}' : \tau_h$. Let A be an agent-term such that $\Psi; \emptyset \vdash_a \lambda x_a{:}\mathsf{t}.\, A : \mathsf{t} \to \mathsf{b}$ and further suppose that $(M,\ \lambda x_a{:}\mathsf{t}.\, A)$ is host-free. Then:*

$$(M,\ (\lambda x_a{:}\mathsf{t}.\, A)\llcorner \hat{H} \lrcorner_h^{\mathsf{t}}) \longmapsto^* c$$
$$\text{iff}$$
$$(M,\ (\lambda x_a{:}\mathsf{t}.\, A)\llcorner \hat{H}' \lrcorner_h^{\mathsf{t}}) \longmapsto^* c$$

Proof: After one step for each evaluation, the configurations are in a form so that the Value Abstraction lemma applies. Since that lemma preserves its own preconditions, inductively apply it to the rest of the evaluation. This shows that the two evaluation sequences must be in lock-step, and so halt with the same value, c. $\square$

Lemma 2.15 (Constant Embeddings) *If* $\vdash M : \Psi$ *and* $\Psi; [x_\kappa : \mathsf{b}] \vdash e : \tau$ *then* $(M,\ \{c/x_\kappa\}e) \longmapsto^* (M',\ \hat{e})$ *if and only if* $(M,\ \{\llcorner c \lrcorner^{\mathsf{b}}_{\kappa^\circ}/x_\kappa\}e) \longmapsto^* (M',\ \hat{e})$.

Proof (sketch): The proof goes by induction on the derivation of the evaluation sequence. The idea is to simply insert **(AConst)** or **(HConst)** steps whenever the embedding $\llcorner c \lrcorner^{\mathsf{b}}_{\kappa^\circ}$ appears. $\square$

Theorem 2.16 (Host-provided Values) *Suppose* M *is a heap such that* $\vdash M : \Psi$. *Let* $\hat{A} = \lambda y_a{:}\mathsf{b} \to \mathsf{t}.\, A$ *be an agent function such that* $\Psi; \emptyset \vdash_a \hat{A} : \tau$ *and* $(M,\ \hat{A})$ *is host-free. Further suppose that* $\Psi; [x_h : \mathsf{b}] \vdash_h ho : \tau_h$ *and* $(M,\ \lambda x_h{:}\mathsf{b}.\, ho)$ *is agent-free. If* $(M,\ (\lambda y_a{:}\mathsf{b} \to \mathsf{t}.\, A)\ \llcorner \lambda x_h{:}\mathsf{b}.\, ho \lrcorner^{\mathsf{b}\to\mathsf{t}}_h) \longmapsto^* (M',\ A')$ *then for any sub-term* $\llcorner \hat{H} \lrcorner^{\mathsf{t}}_h$ *of* A', *there is a constant* c *and heap* M'' *such that* $(M'',\ \{c/x_h\}ho) \longmapsto^* (M''',\ \hat{H})$.

Proof: After one step of evaluation, the configuration is $(M,\ (\lambda y_a{:}\mathsf{b} \to \mathsf{t}.\, A)\ \lambda x_a{:}\mathsf{b}.\, \llcorner \{\llcorner x_a \lrcorner^{\mathsf{b}}_a/x_h\}ho \lrcorner^{\mathsf{t}}_h)$. The following lemma applies because the only reachable host-embedding is of type t and nearly-agent free (the only freely occurring agent-terms are x_a). The lemma implies that under any sequence of transition steps, the only thing that can happen to a host-embedding is the substitution of a constant for x_a, after which it may take host-transition steps. But this says exactly that there is some memory configuration M'' such that $(M'',\ \{c/x_a\}\{\llcorner x_a \lrcorner^{\mathsf{b}}_a/x_h\}ho) \longmapsto (M''',\ \hat{H})$. Since ho is agent-free, we have $\{c/x_a\}\{\llcorner x_a \lrcorner^{\mathsf{b}}_a/x_h\}ho = \{\llcorner c \lrcorner^{\mathsf{b}}_a/x_h\}ho$. The result follows from the Constant Embeddings lemma. $\square$

Lemma 2.17 (Host-Provided Values (step)) *Suppose* M *is a heap such that* $\vdash M : \Psi$. *Let* A *be a closed agent term such that:*

- *If* $\llcorner H \lrcorner^{\tau}_h$ *is a host-embedding reachable from* $(M,\ A)$ *then* $\tau = \mathsf{t}$ *and* H *is nearly-agent free. That is,* $\Psi; [x_a : \mathsf{b}] \vdash_a \llcorner H \lrcorner^{\mathsf{t}}_h : \mathsf{t}$, *and* x_a *is the only agent-term reachable in* $(M,\ H)$.
- *There are no agent colored labels that are reachable from* $(M,\ A)$ *which point to host values. That is, if* ξ^τ *is an agent term reachable from* $(M,\ A)$ *then* $M(\xi^\tau) = (\hat{A}, a)$ *for some* $\hat{A}$.

Further suppose that $(M,\ A) \longmapsto (M',\ A')$. *Then every agent label reachable from* $(M',\ A')$ *points to an agent value, and, in addition, if* $\llcorner H' \lrcorner^{\tau}_h$ *is any host-embedding reachable from* $(M',\ A')$ *then* $\tau = \mathsf{t}$ *and one of the following is true:*

1. $\llcorner H' \lrcorner^{\mathsf{t}}_h$ *is reachable from* $(M,\ A)$.
2. $H' = \{c/x_a\}H$ *where* $\llcorner H \lrcorner^{\mathsf{t}}_h$ *is reachable from* $(M,\ A)$.
3. *There is some* $\llcorner H \lrcorner^{\mathsf{t}}_h$ *reachable from* $(M,\ A)$ *and* $(M,\ H) \longmapsto (M',\ H')$ *and* H' *is agent-free.*

Proof: Proof by induction on the derivation that $(M,\ A) \longmapsto (M',\ A')$. We proceed by cases:

(P1) Here, A is of the form $A^1\ A^2$. From the assumptions, it follows that the only host-embeddings reachable from $(M,\ A^1)$ also satisfy the requirements for the lemma. We also know that $(M,\ A^1) \longmapsto (M',\ A^3)$ for some A^3 and that $A' = A^3\ A^2$. Let $\llcorner H' \lrcorner^{\tau}_h$ be a host-embedding reachable from $(M',\ A')$ and suppose it is not reachable from $(M,\ A)$. By the lemma below, $\llcorner H' \lrcorner^{\tau}_h$ must be reachable from $(M',\ A^3)$ but not $(M,\ A)$.

Lemma 2.18 *The set of reachable host-embeddings for* $(M',\ A^2)$ *is also reachable from* $(M,\ A)$.

Proof: If an embedding is reachable from A^2 it is either a subterm of A^2 or reached via some reference in the heap. Since A^2 doesn't change, the only embeddings that could be affected by the transition step must have been reached via reference shared with the term A^1, which took the transition step. The heap can change in only two ways: by assignment via rule **(AAssn)** or **(HAssn)**, in which case the subterm $\xi^\tau := \hat{e}$ must occur in A^1 and hence be reachable from $(M,\ A)$, or by creation of a new reference (**(Aref)** or **(Href)**) which cannot be referred to by A^2 because the newly created label is fresh. □

The embedding part of the conclusion then follows from the inductive hypothesis, since any new host-embedding must fall into cases (2) or (3).

Now we must show that any agent-colored label reachable from $(M',\ A')$ points to an agent value in the heap. By the inductive hypothesis all agent labels reachable from $(M',\ A^3)$ must point to agent values, so let ξ^τ be an agent label reachable from $(M',\ A^2)$. If ξ^τ is also reachable from $(M',\ A^3)$ we're done, so assume it isn't. If ξ^τ wasn't reachable from $(M,\ A^1)$ then the transition step could not affect it and hence, by assumption it must point to an agent value. If ξ^τ was reachable from $(M,\ A^1)$ then there are two possible ways the value may have changed: If **(AAssn)** was used, then the label still points to an agent value. The other possibility was that **(HAssn)** was used, *but this can't happen.* Why? If **(HAssn)** was applied to ξ^τ then the term $H = \xi^\tau := \hat{H}$ appears as a subterm of A, but then, since ξ^τ pointed to an agent value, the host term, which must be in an embedding, is not nearly-agent free. This contradicts our assumptions about host-embeddings reachable from $(M,\ A)$. Thus we conclude that any agent label reachable from $(M',\ A')$ must point to an agent value.

(P2), (PAssn1), (PAssn2) These cases follow analogously to **(P1)**.

(Pβ) In this case, $A = (\lambda y_a{:}\tau'.\, A^1)\ \hat{A}$. The heap doesn't change, so let $\llcorner H' \lrcorner_h^\tau$ be a term reachable from $(M,\ \{\hat{A}/y_a\}A^1)$ and suppose it is not reachable from $(M,\ A)$. Since the terms changed affected by the substitution $\{\hat{A}/y_a\}$ are exactly those containing free occurrences of y_a, any host-embeddings not in A must be of the form $\{\hat{A}/y_a\}\llcorner H \lrcorner_h^{\mathtt{t}}$. By assumption, we have $\Psi; [x_a : \mathtt{b}] \vdash_a \llcorner H \lrcorner_h^{\mathtt{t}} : \mathtt{t}$, and it follows that $y_a = x_a$ and consequently, by the Canonical Forms Lemma, $\hat{A} = c$ for some constant c. Thus the new host-term is $\{c/x_a\}\llcorner H \lrcorner_h^{\mathtt{t}} = \llcorner \{c/x_a\}H \lrcorner_h^{\mathtt{t}}$ and hence clause 2 of the result is satisfied. Since the heap doesn't change and labels reachable from $(M',\ A')$ are a subset of those reachable from $(M,\ A)$, the condition on agent-colored labels is also satisfied.

(Pμ1) In this case we have $A = \mathtt{roll}_\tau\ A^1$ and that $(M,\ A^1) \longmapsto (M',\ A^2)$. Since A^1 is a subterm of A and A isn't an embedding, it follows that any host-embeddings reachable from $(M,\ A^1)$ are also reachable from $(M,\ A)$. The result follows trivially from the inductive hypothesis.

(Pμ2), (P!), (PRef1) These follow analogously to the case for **(Pμ1)**.

(Pμ3) The $A = \mathtt{unroll}_{\mu\alpha.\tau}\ \mathtt{roll}_{\mu\alpha.\tau}\ A^1$. Since the heap doesn't change and all subterms of A^1 are subterms of A, any host-embedding of A^1 satisfies clause 1, while agent-colored labels still point to the same things.

(AH) Then $A = \llcorner H \lrcorner_h^{\mathtt{t}}$. Furthermore, $(M,\ H) \longmapsto (M',\ H')$. Since $(M,\ H)$ is closed and nearly-agent free, it follows that $(M,\ H)$ is agent-free. A simple argument by induction shows that agent- or host-freeness is preserved by transition steps. Thus we conclude H' is agent-free and so clause 3 is satisfied. Since H is agent-free, any labels contained in H must point to host values and not agent values. Thus, even if H transitions via **(HAssn)**, only locations that are already host-colored are affected. This suffices to show that the agent-colored label property still holds.

(AConst), (AHfn), (Aμ1), and (ALab) All of these cases are eliminated since all embedded host-terms that appear in A must have label $\mathtt{t}$.

(ARef) Any host-embedding that is reachable from $(M[\xi^\tau \mapsto (\hat{A}, a)],\ \xi^\tau)$ is also reachable from $(M,\ \mathtt{ref}_\tau \hat{A})$ and so satisfies clause 1. The newly created label refers to an agent location, and by assumption all agent labels reachable from $(M,\ \hat{A})$ also point to agent values, so that part of the lemma is satisfied.

$$
\begin{array}{rrcl}
\text{(types)} & \tau & ::= & t \mid \mathsf{b} \mid \tau_1 \to \tau_2 \\[1ex]
\text{(labels)} & \ell_i & ::= & i \mid \ell_i : \ell_j \\[1ex]
(i\text{-terms}) & e_i & ::= & x_i \mid c \mid \lambda x_i{:}\tau.\, e_i \mid e_i\ e_i' \\
 & & & \mid \mathtt{fix}\ f_i(x_i{:}\tau).\, e_i \mid \llcorner e_j \lrcorner^{\tau}_{\ell_j} \\[1ex]
(i\text{-primvals}) & \hat{v}_i & ::= & c \mid \lambda x_i{:}\tau.\, e_i \\
(i\text{-values}) & v_i & ::= & \hat{v}_i \mid \llcorner \hat{v}_j \lrcorner^{t}_{\ell_j}\ (\Delta_i(t) = t)
\end{array}
$$

Figure 5: Multiagent Syntax

(A!) Then $A = !\xi^\tau$ and by assumption, we have $M(\xi^\tau) = (\hat{A}, a)$. From the definition of reachability, it follows that any host-embedding reachable from $(M,\ M(\xi))$ is also reachable from $(M,\ A)$. Thus clause 1 is satisfied. Since any agent label reachable from $(M,\ \hat{A})$ is also reachable from $(M,\ !\xi^\tau)$ the condition on agent labels is satisfied.

(AAssn) As in **(A!)**, all host-embeddings reachable from A' are also reachable from A, satisfying clause 1. Also, since $\hat{A}$ is stored as an agent value and every other agent label reachable from $(M',\ A')$ is reachable from $(M,\ A)$, the agent-label condition is met.

□

3 The Multiagent Language

So far, we have described a two-principal setting in which the host has strictly more information than the agent. Many interesting cases can be modeled in this fashion, but there are times when both principals wish to hide information or there are more than two agents involved. For example, we need a multiagent setting to prove safety properties about nested or mutually recursive abstract datatypes.

Another natural generalization is to allow an agent to export multiple abstract types. Once that has been introduced, agents should be able to share type information.

Generalizing the language these ways has another benefit: we can prove theorems once for a broad class of systems. The type abstraction properties for an instance of the system follow as corollaries.

Just as in the two-agent case it is straightforward (but tedious) to include products, sums and recursive types in the multiagent calculus. We also believe we can add state as in the two-agent case. The key technique is to record which agent last assigned to a location, and do the appropriate embedding on derefernce. However, we do not incorporate state in the development below.

3.1 Syntax

Figure 5 shows the syntax for the multiagent language. The types include a base type, b, function types, and type variables ranged over by t, u, and s.

Rather than just two "colors" of terms, we now assume that there are n agents, where n is fixed. In the syntax, the meta-variables i and j range over $\{1, \ldots, n\}$.

Agent i's terms include variables, x_i, non-recursive functions, $\lambda x_i{:}\tau.\, e_i$, recursive functions, $\mathtt{fix}\, f_i(x_i{:}\tau).\, e_i$, function applications, $e_i\ e_i'$, and embeddings $\llcorner e_j \lrcorner^{\tau}_{\ell_j}$. We include both recursive and non-recursive functions to simplify the dynamic semantics (see rule **(4)** in Figure 6).

An embedding containing a j-term is labeled with a *list* of agents beginning with j for reasons explained in the discussion of the semantics. We use the notation ℓ_j for a non-empty list of agents beginning with j. If ℓ_i and ℓ_j are two such lists, then $\ell_i{:}\ell_j$ is ℓ_i appended to ℓ_j and $\mathtt{rev}(\ell_i)$ is the list-reversal of ℓ_i.

The goal is a language in which each agent has limited knowledge of type information. Thus, we must somehow represent what an agent "knows" and ensure that agents sharing information do so consistently.

$$(1)\ \frac{e_i^0 \overset{i}{\longmapsto} e_i^1}{e_i^0\ e_i^2 \overset{i}{\longmapsto} e_i^1\ e_i^2} \qquad (2)\ \frac{e_i^0 \overset{i}{\longmapsto} e_i^1}{v_i\ e_i^0 \overset{i}{\longmapsto} v_i\ e_i^1} \qquad (3)\ \frac{e_j^0 \overset{j}{\longmapsto} e_j^1}{\llcorner e_j^0 \lrcorner_{\ell_j}^{\tau} \overset{i}{\longmapsto} \llcorner e_j^1 \lrcorner_{\ell_j}^{\tau}}$$

$$\begin{array}{llcll}
(4) & \mathtt{fix}\ f_i(x_i{:}\tau).\, e_i & \overset{i}{\longmapsto} & \lambda x_i{:}\tau.\, \{\mathtt{fix}\ f_i(x_i{:}\tau).\, e_i/f_i\}e_i & \\
(5) & \lambda x_i{:}\tau.\, e_i\ \ v_i & \overset{i}{\longmapsto} & \{v_i/x_i\}e_i & \\
\\
(6) & \llcorner c \lrcorner_{\ell_j}^{\mathsf{b}} & \overset{i}{\longmapsto} & c & \\
(7) & \llcorner \hat{v}_j \lrcorner_{\ell_j}^{\tau} & \overset{i}{\longmapsto} & \llcorner \hat{v}_j \lrcorner_{\ell_j}^{\bar{\Delta}_i(\tau)} & (\tau \neq \Delta_i(\tau)) \\
(8) & \llcorner \llcorner \hat{v}_j \lrcorner_{\ell_j}^{u} \lrcorner_{\ell_k}^{\tau} & \overset{i}{\longmapsto} & \llcorner \hat{v}_j \lrcorner_{\ell_j : \ell_k}^{\tau} & (u \notin \mathrm{Dom}(\Delta_k), \tau = \Delta_i(\tau)) \\
\\
(9) & \llcorner \lambda x_j{:}\tau^0.\, e_j \lrcorner_{\ell_j}^{\tau^1 \to \tau^2} & \overset{i}{\longmapsto} & \lambda x_i{:}\tau^1.\, \llcorner \{\llcorner x_i \lrcorner_{i:\mathtt{rev}(\ell_j)}^{\tau^0}/x_j\}e_j \lrcorner_{\ell_j}^{\tau^2} & (x_i \text{ fresh}, \tau^1 \to \tau^2 = \Delta_i(\tau^1 \to \tau^2))
\end{array}$$

Figure 6: Multiagent Dynamic Semantics: $e_i \overset{i}{\longmapsto} e_i'$

For example, agent i might know that $\mathsf{fh} = int$. Agent j may or may not have this piece of information, but if j does know the realization of fh, that knowledge must be compatible with what i knows. (It shouldn't be the case that j thinks $\mathsf{fh} = string$.)

The consistency of the type information of the language is encapsulated in an "omniscient" map, Φ, that takes type variables to types. It is a finite map of the following form:

$$\Phi ::= \ \emptyset \mid \Phi[t = \mathsf{b}] \mid \Phi[t = s],\quad s \in Dom(\Phi) \mid \Phi[t = s \to u],\quad s,\ u \in Dom(\Phi)$$

Intuitively, Φ represents the "real" type alias information in the system. For instance in the situation above, Φ might be $[\mathsf{fh} = int]$. In practice, Φ could be extracted from the source code of the agents and checked for consistency.

Accordingly, each principal will know some of the information contained in Φ. For each agent, i, let Δ_i be a subset of $Dom(\Phi)$. Each such Δ_i determines a map, $\Phi|_{\Delta_i}$ obtained by restricting the domain of Φ to Δ_i. To simplify notation, we use $\Delta_i(t)$ for $\Phi|_{\Delta_i}(t)$. Continuing the example above, we might have $\Delta_i = \{\mathsf{fh}\}$ and $\Delta_j = \{\}$, which says that i knows that fh is implemented as an integer while j does not. Agent i *knows* t means $t \in \Delta_i$.

Note that since Δ_i and Δ_j are both restrictions of Φ, it is the case that for any $t \in Dom(\Delta_i) \cap Dom(\Delta_j)$ we have $\Delta_i(t) = \Delta_j(t)$. This restriction is not as prohibitive as it may appear. For example, we can encode the maps $\Delta_i(\mathsf{t}) = \mathsf{s}$, $\Delta_i(\mathsf{s}) = int$, $\Delta_j(\mathsf{t}) = int$ by rearranging the functions to be $\Delta_i(\mathsf{s}) = \mathsf{t}$, $\Delta_i(\mathsf{t}) = \Delta_j(\mathsf{t}) = int$. This does not work in all cases, however, but it is sufficient to ensure Type Relations property (*iii*) (see Lemma 4.5)[1].

Each Δ_i extends naturally to a map from an arbitrary type τ as follows:

$$\begin{array}{rcl}
\Delta_i(\mathsf{b}) & = & \mathsf{b} \\
\Delta_i(t) & = & \begin{cases} t & t \notin \mathrm{Dom}(\Delta_i) \\ \Delta_i(t) & t \in \mathrm{Dom}(\Delta_i) \end{cases} \\
\Delta_i(\tau^1 \to \tau^2) & = & \Delta_i(\tau^1) \to \Delta_i(\tau^2)
\end{array}$$

Such a Δ_i determines a complete partial order, $\sqsubseteq_{\Delta_i}$, on types. We denote the least upper bound of the sequence $\tau \sqsubseteq_{\Delta_i} \Delta_i(\tau) \sqsubseteq_{\Delta_i} \Delta_i^2(\tau) \sqsubseteq_{\Delta_i} \ldots$ by $\bar{\Delta}_i(\tau)$. Thus, $\bar{\Delta}_i(\tau)$ is the most concrete view of τ that agent i is able to determine from its knowledge. For succinctness, we write $\{\Delta\}$ for the set $\{\Delta_1, \ldots, \Delta_n\}$.

Some key properties of these Δ maps are summarized below:

[1] In fact, resorting to this restrictive "omniscient" Φ is unnecessary. It is sufficient for our system to have a set of *compatible* Δ maps, where compatible is taken to mean that no collection of agents can relate (in the sense of Figure 8) two incompatible types (i.e. a base type and an arrow type). The Δ maps must further satisfy Lemma 3.7. The requirements on the Δ maps given here are sufficient, but not necessary to meet these requirements and can probably be considerably relaxed.

(**const**) $$\{\Delta\};\Gamma \vdash_i c : \mathsf{b}$$

(**var**) $$\{\Delta\};\Gamma \vdash_i x_i : \Gamma(x_i)$$

(**app**) $$\frac{\{\Delta\};\Gamma \vdash_i e_i : \tau' \to \tau \qquad \{\Delta\};\Gamma \vdash_i e'_i : \tau'}{\{\Delta\};\Gamma \vdash_i e_i\ e'_i : \tau}$$

(**abs**) $$\frac{\{\Delta\};\Gamma[x_i : \tau'] \vdash_i e'_i : \tau \qquad \Delta_i(\tau') = \tau'}{\{\Delta\};\Gamma \vdash_i \lambda x_i{:}\tau'.\, e'_i : \tau' \to \tau}(x_i \notin \mathrm{Dom}(\Gamma))$$

(**fix**) $$\frac{\{\Delta\};\Gamma[f_i : \tau' \to \tau][x_i : \tau'] \vdash_i e'_i : \tau \qquad \Delta_i(\tau' \to \tau) = \tau' \to \tau}{\{\Delta\};\Gamma \vdash_i \mathtt{fix}\ f_i(x_i{:}\tau').\, e'_i : \tau' \to \tau}(f_i, x_i \notin \mathrm{Dom}(\Gamma))$$

(**embed**) $$\frac{\{\Delta\};\Gamma \vdash_j e_j : \tau' \qquad \{\Delta\} \vdash \tau' \lesssim_{\ell_j:i} \tau}{\{\Delta\};\Gamma \vdash_i \llcorner e_j \lrcorner^{\tau}_{\ell_j} : \bar{\Delta}_i(\tau)}$$

Figure 7: Multiagent Static Semantics: $\{\Delta\};\Gamma \vdash_i e_i : \tau$

Lemma 3.1 (Consistency) *For any agents i and j and type variable t, if $t \in \Delta_i \cap \Delta_j$ then $\Delta_i(t) = \Delta_j(t)$.*

Proof: Immediate from the definition of Δ_i and Δ_j. □

Note that $\Delta_i(\Delta_j(\tau))$ is not necessarily $\Delta_j(\Delta_i(\tau))$.

Lemma 3.2 (Properties of $\bar{\Delta}_i$) *For all agents i and types τ and τ',*

(i) $\bar{\Delta}_i(\mathsf{b}) = \mathsf{b}$

(ii) $\bar{\Delta}_i(\tau) = \bar{\Delta}_i(\Delta_i(\tau)) = \Delta_i(\bar{\Delta}_i(\tau))$

(iii) $\bar{\Delta}_i(\tau \to \tau') = \bar{\Delta}_i(\tau) \to \bar{\Delta}_i(\tau')$

(iv) $\bar{\Delta}_i(\tau \to \tau') = \tau^0 \to \tau^1$ *for some τ^0 and τ^1. In particular $\bar{\Delta}_i(\tau \to \tau') \neq \mathsf{b}$*

(v) $\bar{\Delta}_i(\tau) = \tau \iff \Delta_i(\tau) = \tau$

Proof: Immediate from the definition of $\bar{\Delta}_i$. □

The set of i-terms that are values depends on i's available type information. In addition to the usual notion of values, given by i-primvals, a j-primval embedded in agent i is a value if i cannot determine any more type information about the value, *i.e.* $\llcorner \hat{v}_j \lrcorner^{t}_{\ell_j}$ is a value if $\Delta_i(t) = t$.

3.2 Dynamic Semantics

Figure 6 shows the operational semantics for agent i in the multiagent language.

Rules **(1)**, **(2)**, **(4)** and **(5)** establish a typical call-by-value semantics. Rule **(3)** allows evaluation inside embeddings and distinguishes i transitions from j transitions.

Rule **(6)** lets agent i pull a constant, exported at base type, out of an embedding. It corresponds to rules **(A2)** and **(H2)** of the two-agent scenario.

As in the two-agent case where the host had more type information than the agent, an agent can use its knowledge to refine the type of an embedded term. Previously, the substitution $\{\tau_h/\mathsf{t}\}$ in rule **(H3)** served this purpose. Now, $\bar{\Delta}_i$ captures the type refinement information available to agent i. Correspondingly, rule **(7)** allows i to refine the type of an embedded value.

Perhaps the most subtle issue is when to allow embeddings to be stripped away. In the two-agent case, rule **(H4)** let the host pull out its own value that had been embedded abstractly inside an agent. This was safe because the agent had strictly less information than the host. Now, however, an intermediary agent with more knowledge could contribute to the evaluation of a term. If we throw away that information by simply discarding the intermediary's embedding brackets, it becomes difficult to track the relationship between the type of the term inside the embedding and the annotation on the embedding.

Thus we use lists of agents as the labels on embeddings. Intuitively, an agent "signs" the term if it participated in the evaluation. Rule **(8)** says that if there are nested embeddings, $\llcorner\llcorner\hat{v}_j\lrcorner^{u}_{\ell_j}\lrcorner^{\tau}_{\ell_k}$, and the inner embedding, $\llcorner\hat{v}_j\lrcorner^{u}_{\ell_j}$, is a k-value (that is, $u \notin \mathrm{Dom}(\Delta_k)$), then the two embeddings can be collapsed into one, $\llcorner\hat{v}_j\lrcorner^{\tau}_{\ell_j:\ell_k}$. We lose no information about which agents have participated in the evaluation of the term, because we append the two lists. The idea, formalized in the next section, is that the type of the term inside an embedding is related, via the list of agents labeling the embedding, to the type annotation.

The most interesting rule is **(9)**, which is really what tracks the principals. The embedded function is lifted to the outside. Its argument now belongs to the outer agent, i, instead of the inside agent, j. As such, it must be given the type that i thinks the argument should have. The body of the function is still a j-term embedded in an i-term so any occurrence of the new formal argument x_i must be embedded as an i-term inside a j-term. The corresponding type annotation must be the type which j expects the argument to have. Hence the function body is still abstract to i and when the function is applied, the actual argument will be held abstract from j. The only remaining issue is the agent list on the formal argument embeddings. Since the "inside type" and "outside type" have reversed roles, the agent-list must be in reverse order. Intuitively, the agents which successively provided the function argument type to i must undo their work in the body of the function which is a j-term.

3.3 Static Semantics

Figure 7 shows the multiagent static semantics. The rules are parameterized by the agent i. The judgment $\{\Delta\}; \Gamma \vdash_i e_i : \tau$ should be read as, "Under type-maps $\Delta_1, \ldots, \Delta_n$ in context Γ, agent i can show that e_i has type τ".

All of the rules except **(embed)** are essentially standard. The rules **(abs)** and **(fix)** have additional conditions that force an agent to use the most concrete type available for functions internal to the agent.

As alluded to previously, the issue of consistency between agents arises during type checking. For instance, we don't want an agent to export an *int* as a function. Likewise, we don't want an agent, or collection of agents, to violate the type abstractions represented by the Δ_i's. Thus we need some way of relating the type of the expression inside the embedding to the typing annotation on the embedding.

We establish an agent-list indexed family of relations on types, $\tau \lesssim_{\ell_i} \tau'$. Judgments of the form $\{\Delta\} \vdash \tau \lesssim_{\ell_i} \tau'$, showing when two types may be related by the list ℓ_i, are given in Figure 8. These rules say that $\tau^0 \lesssim_{i_1:i_2:\ldots:i_n} \tau^n$ if there exist types $\tau^1, \ldots, \tau^{n-1}$ such that agent i_k is able to show that $\tau^{k-1} = \tau^k$ for $k \in \{1, \ldots, n\}$. Informally, the agents are able to chain together their knowledge of type information to show that $\tau^0 = \tau^n$.

The **(embed)** rule uses the $\lesssim_{\ell_j:i}$ relation to ensure that the type inside the embedding matches up with the annotation on the embedding. The agent i is appended to the list because, as the outermost agent, i is implicitly involved in evaluation of the term.

Why is this somewhat complicated mechanism necessary? To some extent, it's not. It is clear that there must be some way of relating the type of a term inside an embedding to the type annotation on the embedding; otherwise, for example, an agent could export an integer as a string. We could have chosen to allow nested embeddings to be values, so long as each inner embedding was a value with respect to the enclosing agent (for example $\llcorner\llcorner 3\lrcorner^{\mathsf{t}}_{i}\lrcorner^{\mathsf{s}}_{j}$ would be a k-value if $\mathsf{s} \notin \mathrm{Dom}(\Delta_k)$ and $\mathsf{t} \notin \mathrm{Dom}(\Delta_j)$). This allows embeddings to "pile up" in a way that is difficult to deal with syntactically and that complicates the dynamic semantics.

Instead, we allow rule **(8)** to collapse two embeddings and push the work of ensuring compatibility onto the $\lesssim_{\ell_i}$ relation. The lists contain all of the agents that have participated in the evaluation of the inner value because inconsistencies might arise otherwise. Consider three agents, i, j, and k such that $\Delta_i(\mathsf{t}) = int$, $\Delta_j(\mathsf{s}) = \mathsf{t}$ and $\Delta_k = \emptyset$. Then collapsing the properly typed k-term $\llcorner\llcorner 3\lrcorner^{\mathsf{t}}_{i}\lrcorner^{\mathsf{s}}_{j}$ to either $\llcorner 3\lrcorner^{\mathsf{s}}_{i}$ or $\llcorner 3\lrcorner^{\mathsf{s}}_{j}$ violates the type abstraction properties since neither i nor j knows that s abstracts an *int*. Alternately, if we were to use sets of agents, instead of (ordered) lists, the reasonable typing rules become too permissive.

$$\textbf{(eq)} \quad \frac{\bar{\Delta}_i(\tau) = \bar{\Delta}_i(\tau')}{\{\Delta\} \vdash \tau \lesssim_i \tau'}$$

$$\textbf{(trans)} \quad \frac{\{\Delta\} \vdash \tau' \lesssim_{\ell_i} \tau'' \quad \{\Delta\} \vdash \tau'' \lesssim_{\ell_j} \tau'}{\{\Delta\} \vdash \tau \lesssim_{\ell_i:\ell_j} \tau'}$$

Figure 8: Type relations: $\{\Delta\} \vdash \tau \lesssim_{\ell_i} \tau'$

3.4 Type Soundness

We first need to prove several properties about related types. We can then follow a traditional approach to syntactic type soundness.

3.4.1 Type Relation Lemmas

Given the collection of Δ's, we can construct a graph, with edges colored by agents, that captures the relation between types. In particular, let $G(\{\Delta\}) = (V, E)$ where V, the vertices, is just the set of all types and E is a set of colored edges. For each agent i and types τ and τ' there is an edge $(\tau, \tau')_i \in E$ if and only if either $\tau = \tau'$, $\Delta_i(\tau) = \tau'$, or $\Delta_i(\tau') = \tau$. These edges are undirected, and the notation $(\tau, \tau')_i$ indicates that there is an i-colored edge from τ to τ'. Wherever $\{\Delta\}$ is clear from the context or unimportant, we write G instead of $G(\{\Delta\})$.

Given such a graph, G, a *path* of length k in the graph from τ to τ' is a sequence of k edges in E $(\tau^0, \tau^1)_{i_0}$, $(\tau^1, \tau^2)_{i_1}, \ldots, (\tau^{k-1}, \tau^k)_{i_{k-1}}$ where $\tau^0 = \tau$ and $\tau^k = \tau'$ and i_m is an agent for each m. Since we will be concerned with paths with particular properties, we also define the notion of an $i_0^{x_0}, i_1^{x_1}, \ldots, i_k^{x_k}$*-path* from τ to τ'. If $x_m \geq 0$ is an integer for each $m \in \{1 \ldots k\}$, and i_m is an agent for each $m \in \{1 \ldots k\}$ then an $i_0^{x_0}, i_1^{x_1}, \ldots, i_k^{x_k}$-path from τ to τ' is a path from τ to τ' such that the first x_0 edges are labeled by agent i_0, the next x_1 edges are labeled i_1, etc. Similarly, we define a *simple path labeled* i in G to be a path where all the edges are labeled by the single agent i, in other words, an i^x-path.

Lemma 3.3 (Reflexivity of $\lesssim$) *For every τ and ℓ_i, $\{\Delta\} \vdash \tau \lesssim_{\ell_i} \tau$.*

Proof: Proof by induction on the length of ℓ_i. If $|\ell_i| = 1$ then $\ell_i = i$. By definition, $\bar{\Delta}_i(\tau) = \bar{\Delta}_i(\tau)$, so by the **(eq)** rule, we have $\{\Delta\} \vdash \tau \lesssim_i \tau$. In the inductive case, $|\ell_i| > 1$, so $\ell_i = i, \ell_j$. By the inductive hypothesis, $\{\Delta\} \vdash \tau \lesssim_{\ell_j} \tau$ and $\{\Delta\} \vdash \tau \lesssim_i \tau$. By the **(trans)** rule (instantiated with τ), the result follows. □

Lemma 3.4 (Idempotency in Lists) *For all τ, τ', agents i, and (possibly empty) lists ℓ and ℓ':*

i $\{\Delta\} \vdash \tau \lesssim_{i:i} \tau'$ *iff* $\{\Delta\} \vdash \tau \lesssim_i \tau'$.

ii For all agents lists of the form $\ell{:}i{:}i{:}\ell'$, $\{\Delta\} \vdash \tau \lesssim_{\ell:i:i:\ell'} \tau'$ iff $\{\Delta\} \vdash \tau \lesssim_{\ell:i:\ell'} \tau'$

Proof: Part (i): Assume $\{\Delta\} \vdash \tau \lesssim_{i:i} \tau'$ then there must exist a τ'' such that $\{\Delta\} \vdash \tau \lesssim_i \tau''$ and $\{\Delta\} \vdash \tau'' \lesssim_i \tau'$. These can be derived only by using the **(eq)** rule, so we have $\bar{\Delta}_i(\tau) = \bar{\Delta}_i(\tau'') = \bar{\Delta}_i(\tau)$, so by another application of **(eq)**, $\{\Delta\} \vdash \tau \lesssim_i \tau'$. Conversely, assume $\{\Delta\} \vdash \tau \lesssim_i \tau'$. Then $\bar{\Delta}_i(\tau) = \bar{\Delta}_i(\tau')$. It follows by the **(trans)** rule by instantiating τ'' with τ that $\{\Delta\} \vdash \tau \lesssim_{i:i} \tau'$. Part (ii): By induction on $|\ell| + |\ell'|$. The base case, 0, is part (i). For the inductive case, in either direction, the derivation must end with **(trans)**. If this rule "splits" the agent list anywhere but "between $i : i$", then the result is immediate from the inductive hypothesis and **(trans)**. So consider the case where the "split" for the forward direction of the iff is $\ell : i$ and $i : \ell'$. It can be shown by induction on the derivation that there must exist some τ^3 such that $\{\Delta\} \vdash \tau'' \lesssim_i \tau^3$ and $\{\Delta\} \vdash \tau^3 \lesssim_{\ell'} \tau'$. This allows us to "move the split" by using **(trans)** with $\ell : i$ and i, then a second time with ℓ', reducing us to a previous case. □

Lemma 3.5 (Simple Paths) *If $\bar{\Delta}_i(\tau) = \tau'$ then there is a simple path labeled i from τ to τ' in G.*

Proof: From the definition of $\bar{\Delta}_i(\tau)$ there exists a k such that $\Delta_i^k(\tau) = \tau'$. By induction on k it is easy to construct a simple path of length k labeled i in G. □

Lemma 3.6 (Type Relations are Paths) *Let $\{\Delta\}$ be a consistent set of type maps and let $\ell_i = i_0 : i_1 : \ldots : i_k$, where $i_0 = i$, be a list of agents. Then for all τ and τ', $\{\Delta\} \vdash \tau \lesssim_{\ell_i} \tau'$ if and only if there are non-negative integers $x_0, x_1, \ldots, x_k$ such that there is an $i_0^{x_0}, i_1^{x_1}, \ldots, i_k^{x_k}$- path from τ to τ' in $G(\{\Delta\})$.*

Proof: Assume that $\{\Delta\} \vdash \tau \lesssim_{\ell_i} \tau'$. We proceed by induction on the derivation of this fact. If the last step in the derivation was **(eq)** then $\ell_i = i$ and we have $\bar{\Delta}_i(\tau) = \bar{\Delta}_i(\tau')$. From the Simple Paths lemma, there is an i^n-path from τ to $\bar{\Delta}_i(\tau)$ and an i^m-path from τ' to $\bar{\Delta}_i(\tau')$. Since they are equal, there is an i^{n+m}-path from τ to τ'. If the last step of the derivation was **(trans)** then $\ell_i = \ell_i' : \ell_j$ for some $\ell_i' = i_0 : i_1 : \ldots, i_m$ and $\ell_j = i_{m+1} : \ldots : i_k$ and there exists a τ'' such that $\{\Delta\} \vdash \tau \lesssim_{\ell_i'} \tau''$ and $\{\Delta\} \vdash \tau'' \lesssim_{\ell_j} \tau'$. By the inductive hypothesis, there is an $i_0^{x_0}, i_1^{x_1}, \ldots, i_m^{x_m}$-path from τ to τ'' and an $i_{m+1}^{x_{m+1}}, \ldots, i_k^{x_k}$-path from τ'' to τ'. By concatenating these paths, we get an $i_0^{x_0}, i_1^{x_1}, \ldots, i_k^{x_k}$-path from τ to τ'.

Conversely, assume there is an $i_0^{x_0}, i_1^{x_1}, \ldots, i_k^{x_k}$-path from τ to τ'. We prove by induction on the length of the path, $len = \sum_{m=0}^{k} x_m$, that $\{\Delta\} \vdash \tau \lesssim_{\ell_i} \tau'$. For the base case, $len = 0$. Then it must be that $\tau = \tau'$. So by reflexivity of $\lesssim$, we are done. For the inductive case, $len > 0$. Let e be the first edge in the path. It belongs to some agent i_m and connects τ with some τ''. By the definition of $G(\{\Delta\})$ and Lemma 3.2(ii), $\bar{\Delta}_{i_m}(\tau) = \bar{\Delta}_{i_m}(\tau'')$. So by **(eq)**, $\tau \lesssim_{i_m} \tau''$. By reflexivity of $\lesssim$, we know $\tau \lesssim_{i_0:\ldots:i_{m-1}} \tau$. Furthermore, there is an $i_m^{x_m - 1}, i_{m+1}, \ldots i_k$-path from τ'' to τ'. So by the inductive hypothesis, $\tau'' \lesssim_{i_m:\ldots:i_k} \tau'$. By two invocations of **(trans)**, $\tau \lesssim_{i_0:\ldots:i_m:i_m:\ldots:i_k} \tau'$. By idempotency, we are done.

□

Lemma 3.7 (Type Relations) *For all agents i and j and types τ^0, τ^1, τ^2 and τ^3:*

i If $\{\Delta\} \vdash \bar{\Delta}_i(\tau^0) \lesssim_{\ell_i} \tau^1$ then $\{\Delta\} \vdash \tau^0 \lesssim_{\ell_i} \tau^1$.

ii If $\{\Delta\} \vdash \tau^0 \lesssim_{\ell_i} \tau^1$ then $\{\Delta\} \vdash \tau^1 \lesssim_{\mathtt{rev}(\ell_i)} \tau^0$ where $\mathtt{rev}(\ell_i)$ means reverse(ℓ_i).

iii If $\{\Delta\} \vdash \tau^0 \to \tau^1 \lesssim_{\ell_i} \tau^2 \to \tau^3$ then $\{\Delta\} \vdash \tau^0 \lesssim_{\ell_i} \tau^2$ and $\{\Delta\} \vdash \tau^1 \lesssim_{\ell_i} \tau^3$.

Proof: Unless mentioned otherwise, the proof in each case proceeds by induction on the derivation that $\{\Delta\} \vdash \tau \lesssim_{\ell_i} \tau'$.

i If the last rule in the derivation was (**eq**) then $\ell_i = i$ and $\bar{\Delta}_i(\bar{\Delta}_i(\tau^0)) = \bar{\Delta}_i(\tau^1)$, but since $\bar{\Delta}_i(\tau^0) = \bar{\Delta}_i(\bar{\Delta}_i(\tau^0))$, we also conclude that $\bar{\Delta}_i(\tau^0) = \bar{\Delta}_i(\tau^1)$. The result follows via the (**eq**) rule. Otherwise, the last rule must have been (**trans**), so $\ell_i = \ell_i'$:ℓ_j for some ℓ_i' and ℓ_j and there exists a τ'' such that $\{\Delta\} \vdash \bar{\Delta}_i(\tau^0) \lesssim_{\ell_i'} \tau''$ and $\{\Delta\} \vdash \tau'' \lesssim_{\ell_j} \tau^1$. By the inductive hypothesis, we have $\{\Delta\} \vdash \tau^0 \lesssim_{\ell_i'} \tau''$ so by an application of the **(trans)** rule, the result follows.

ii If the last rule of the derivation was **(eq)** then $\ell_i = i$ and $\bar{\Delta}_i(\tau^0) = \bar{\Delta}_i(\tau^1)$ Since $\mathtt{rev}(\ell_i) = \mathtt{rev}(i) = i$, the result holds by another application of **(eq)**. Otherwise the last rule in the derivation is **(trans)**, in which case $\ell_i = \ell_i'$:ℓ_j and there exists a τ'' such that $\{\Delta\} \vdash \tau^0 \lesssim_{\ell_i'} \tau''$ and $\{\Delta\} \vdash \tau'' \lesssim_{\ell_j} \tau^1$. Two applications of the inductive hypothesis yield $\{\Delta\} \vdash \tau^1 \lesssim_{\mathtt{rev}(\ell_j)} \tau'$ and $\{\Delta\} \vdash \tau' \lesssim_{\mathtt{rev}(\ell_i')} \tau^0$. Since $\mathtt{rev}(\ell_i) = \mathtt{rev}(\ell_i' : \ell_j) = \mathtt{rev}(\ell_j), \mathtt{rev}(\ell_i')$, by the **(trans)** rule we have the desired result.

iii Assume $\{\Delta\} \vdash \tau^0 \to \tau^1 \lesssim_{\ell_i} \tau^2 \to \tau^3$. If the last rule used in the derivation was **(eq)** then $\ell_i = i$ and we have $\bar{\Delta}_i(\tau^0 \to \tau^1) = \bar{\Delta}_i(\tau^2 \to \tau^3)$. It follows from the definition of $\bar{\Delta}_i$ that $\bar{\Delta}_i(\tau^0) = \bar{\Delta}_i(\tau^2)$ and $\bar{\Delta}_i(\tau^1) = \bar{\Delta}_i(\tau^3)$. Thus by two applications of **(eq)**, the result follows. Otherwise the last rule used in the derivation was **(trans)**, in which case $\ell_i = \ell_i' : \ell_j$ from some ℓ_i' and ℓ_j. Also, there exists a type τ such that $\{\Delta\} \vdash \tau^0 \to \tau^1 \lesssim_{\ell_i'} \tau$ and $\{\Delta\} \vdash \tau \lesssim_{\ell_j} \tau^2 \to \tau^3$. There are two cases, depending on whether $\tau = \tau' \to \tau''$ or $\tau = t$.

In the first case, the inductive hypothesis immediately yields $\{\Delta\} \vdash \tau^0 \lesssim_{\ell_i'} \tau'$ and $\{\Delta\} \vdash \tau' \lesssim_{\ell_j} \tau^2$, so by using the **(trans)** rule, we conclude $\{\Delta\} \vdash \tau^0 \lesssim_{\ell_i':\ell_j} \tau^2$. Similar reasoning shows that $\{\Delta\} \vdash \tau^1 \lesssim_{\ell_i':\ell_j} \tau^3$.

We now show that the second case need not occur. It suffices to show that if there is an $i_0^{x_0}, \ldots, i_m^{x_m}$-path from $\tau^0 \to \tau^1$ to $\tau^2 \to \tau^3$, then there exists an $i_0^{x_0}, \ldots, i_m^{x_m}$-path from $\tau^0 \to \tau^1$ to $\tau^2 \to \tau^3$ that has *no variable nodes*. This is sufficient: If the new "no-variable" path has no nodes other than $\tau^0 \to \tau^1$ and $\tau^2 \to \tau^3$ then there is a one-edge path and we can just use **(eq)**. Else there is an intermediate node $\tau^4 \to \tau^5$ and we split the path into "before" and "after" the first time this node is reached. By Relations are Paths, we are now in the first case.

Now suppose t is the first variable encountered on the original path. We know that the node in the path before the first t occurrence is some $\tau^6 \to \tau^7$. Also, we know the path must eventually reach another arrow type node (since it ends at one). We show that the next arrow type node reached must be $\tau^6 \to \tau^7$. Assume for contradiction the next arrow type node reached is different. Then there is a variable-only path in $G(\{\Delta\})$ from t to some other arrow type node. Then there is a cycle-free path to the other node. This path must have interior nodes which are distinct variables $t_1, \ldots, t_n$. But the definition of $G(\{\Delta\})$ and consistency demand that $\Phi(t_1) = t$ since $\Phi(t) = \tau^6 \to \tau^7$. Similarly, it must be that $\Phi(t_2) = t_1$ and so on inductively. But then $\Phi(t_n) = t_{n-1}$. This means an edge from t_n to an arrow node must violate consistency. Contradiction.

Since the path must cycle from $\tau^6 \to \tau^7$, we can eliminate all edges in-between (and just take the self-loop instead). Hence we can remove the first occurrence of a variable node from the path. Inductively, we can remove all variables.

□

3.4.2 Preservation and Progress

Lemma 3.8 (Canonical Forms) *If $\{\Delta\}; \emptyset \vdash_i v_i : \tau$ then:*

- *If $\tau = \mathsf{b}$, then $v_i = c$.*
- *If $\tau = \tau^1 \to \tau^2$, then $v_i = \lambda x_i{:}\tau^1.\, e_i$ for some x_i and e_i.*
- *If $\tau = t$, then $t \notin \Delta_i$ and $v_i = \llcorner \hat{v}_j \lrcorner^t_{\ell_j}$ for some $\hat{v}_j$ and ℓ_j.*

Proof: By inspection of the static semantics, the form of values, and Lemma 3.2. □

Lemma 3.9 (Concreteness of Types) *For any i, if $\{\Delta\}; \Gamma \vdash_i e_i : \tau$ then $\tau = \Delta_i(\tau)$.*

Proof: Easily shown induction on the derivation that $\{\Delta\}; \Gamma \vdash_i e_i : \tau$, using Lemma 3.2 as necessary.
□

Lemma 3.10 (Substitution) *For all agents i and j, if $\{\Delta\}; \Gamma[x_j : \tau'] \vdash_i e_i : \tau$ and $\{\Delta\}; \Gamma \vdash_j e_j : \tau'$, then $\{\Delta\}; \Gamma \vdash_i \{e_j/x_j\}e_i : \tau$.*

Proof: By induction on the typing derivation for $\{\Delta\}; \Gamma[x_j : \tau'] \vdash_i e_i : \tau$. There are several cases, depending on the last rule applied in the derivation:

(const) Immediate since substitution has no effect and $x_j \notin FV(c)$.

(var) Then $e_i = x_i$ for some x_i and there are two possibilities:

$x_i \neq x_j$ Then $\{e_j/x_j\}x_i = x_i$ and by strengthening (since $x_j \notin FV(x_i)$) and the assumptions, $\{\Delta\}; \Gamma \vdash_i x_i : \tau$.

$x_i = x_j$ Then $i = j$ and $\{e_j/x_j\}x_j = e_j$. By **(var)**, $\tau' = \tau$. The result follows.

(app) Then $e_i = e'_i\ e''_i$ for some e'_i and e''_i and $\{\Delta\}; \Gamma[x_j : \tau'] \vdash_i e'_i : \tau'' \to \tau$ and $\{\Delta\}; \Gamma[x_j : \tau'] \vdash_i e''_i : \tau'$. By two applications of the inductive hypothesis, we also have $\{\Delta\}; \Gamma \vdash_i \{e_j/x_j\}e'_i : \tau'' \to \tau$ and $\{\Delta\}; \Gamma \vdash_i \{e_j/x_j\}e''_i : \tau''$. The result follows by **(app)** and the definition of substitution.

(abs) Then it must be that $e_i = \lambda x_i{:}\tau^2.\, e'_i$ and $\tau = \tau^2 \to \tau^3$, for some x_i, e'_i, τ^2, and τ^3. Furthermore, it must be the case that $\{\Delta\}; \Gamma[x_j : \tau'][x_i : \tau^2] \vdash_i e'_i : \tau^3$. By the bound variable convention, $x_j \neq x_i$, so $\{e_j/x_j\}\lambda x_i{:}\tau^2.\, e'_i = \lambda x_i{:}\tau^2.\, \{e_j/x_j\}e'_i$. The inductive hypothesis yields $\{\Delta\}; \Gamma[x_i : \tau^2] \vdash_i \{e_j/x_j\}e'_i : \tau^3$, and we conclude via the (**abs**) rule that $\{\Delta\}; \Gamma \vdash_i \lambda x_i{:}\tau^2.\, \{e_j/x_j\}e'_i : \tau^2 \to \tau^3$. The result follows from the definition of substitution.

(fix) Argument is completely analogous to previous case.

(embed) Then $e_i = \llcorner e_k \lrcorner^{\tau^2}_{\ell_k}$ for some k, e_k, ℓ_k and τ^2. Furthermore, it must be the case that $\{\Delta\}; \Gamma[x_j : \tau'] \vdash_k e_k : \tau^3$, $\{\Delta\} \vdash \tau^3 \lesssim_{\ell_k:i} \tau^2$, and $\tau = \bar{\Delta}_i(\tau^2)$ for some τ^3. By the inductive hypothesis, $\{\Delta\}; \Gamma \vdash_k \{e_j/x_j\}e_k : \tau^3$, so by the **(embed)** rule we conclude that $\{\Delta\}; \Gamma \vdash_i \llcorner \{e_j/x_j\}e_k \lrcorner^{\tau^2}_{\ell_k} : \tau$. The result follows from the definition of substitution.

□

Lemma 3.11 (Preservation) *For all agents i, if $\{\Delta\}; \emptyset \vdash_i e_i : \tau$ and $e_i \overset{i}{\longmapsto} e'_i$ then $\{\Delta\}; \emptyset \vdash_i e'_i : \tau$.*

Proof: By induction on the derivation that $e_i \overset{i}{\longmapsto} e'_i$. The interesting cases are **(7)**, **(8)**, and **(9)**. Rules **(1)**, **(2)**, **(3)** follow directly from the inductive hypothesis and the static semantics. Rules **(4)** and **(5)** follow from the Substitution Lemma (which applies because of Concreteness of Types) along with the **(abs)** and **(fix)** rules. Rule **(6)** follows directly from the static semantics. We consider the other cases separately:

(7) Since e_i typechecks, we must have a derivation as follows:

$$\textbf{(embed)}\ \frac{\{\Delta\}; \emptyset \vdash_j \hat{v}_j : \tau^0 \quad \{\Delta\} \vdash \tau^0 \lesssim_{\ell_j:i} u}{\{\Delta\}; \emptyset \vdash_i \llcorner \hat{v}_j \lrcorner^u_{\ell_j} : \bar{\Delta}_i(u)}$$

Since $\bar{\Delta}_i(u)$ equals $\bar{\Delta}_i(\Delta_i(u))$, we have via the (**eq**) rule that $\{\Delta\} \vdash u \lesssim_i \bar{\Delta}_i(u)$. Thus by (**trans**) it follows that $\{\Delta\} \vdash \tau^0 \lesssim_{\ell_j:i:i} \bar{\Delta}_i(u)$ and we can remove the second i by idempotency. We can now derive:

$$\textbf{(embed)}\ \frac{\{\Delta\}; \emptyset \vdash_j \hat{v}_j : \tau^0 \quad \{\Delta\} \vdash \tau^0 \lesssim_{\ell_j:i} \bar{\Delta}_i(u)}{\{\Delta\}; \emptyset \vdash_i \llcorner \hat{v}_j \lrcorner^{\bar{\Delta}_i(u)}_{\ell_j} : \bar{\Delta}_i(u)}$$

The conclusion is the desired result.

(8) Since e_i typechecks, we must have a derivation as follows:

$$\textbf{(embed)}\ \frac{\textbf{(embed)}\ \dfrac{\{\Delta\}; \emptyset \vdash_j \hat{v}_j : \tau^0 \quad \{\Delta\} \vdash \tau^0 \lesssim_{\ell_j,k} u}{\{\Delta\}; \emptyset \vdash_k \llcorner \hat{v}_j \lrcorner^u_{\ell_j} : \bar{\Delta}_k(u)} \qquad \{\Delta\} \vdash \bar{\Delta}_k(u) \lesssim_{\ell_k,i} \tau}{\{\Delta\}; \emptyset \vdash_i \llcorner \llcorner \hat{v}_j \lrcorner^u_{\ell_j} \lrcorner^\tau_{\ell_k} : \bar{\Delta}_i(\tau)}$$

Furthermore, $u \notin \Delta_k$, so $\bar{\Delta}_k(u) = u$. Thus by using the (**trans**) rule, we have $\{\Delta\} \vdash \tau^0 \lesssim_{\ell_j:k:\ell_k:i} \tau$. And since ℓ_k begins with agent k, idempotency proves that $\{\Delta\} \vdash \tau^0 \lesssim_{\ell_j:\ell_k:i} \tau$, which yields the following derivation:

$$\textbf{(embed)}\ \frac{\{\Delta\}; \emptyset \vdash_j \hat{v}_j : \tau^0 \quad \{\Delta\} \vdash \tau^0 \lesssim_{\ell_j:\ell_k:i} \tau}{\{\Delta\}; \emptyset \vdash_i \llcorner \hat{v}_j \lrcorner^\tau_{\ell_j:\ell_k} : \bar{\Delta}_i(\tau)}$$

The conclusion is the desired result.

(9) Since e_i typechecks, we must have a derivation as follows:

$$\textbf{(embed)}\ \dfrac{\textbf{(abs)}\ \dfrac{\{\Delta\};[x_j:\tau^0]\vdash_j e_j:\tau^3 \quad \Delta_j(\tau^0)=\tau^0}{\{\Delta\};\emptyset\vdash_j \lambda x_j{:}\tau^0.\,e_j:\tau^0\to\tau^3} \qquad \{\Delta\}\vdash\tau^0\to\tau^3 \lesssim_{\ell_j:i} \tau^1\to\tau^2}{\{\Delta\};\emptyset\vdash_i \llcorner\lambda x_j{:}\tau^0.\,e_j\lrcorner_{\ell_j}^{\tau^1\to\tau^2} : \bar{\Delta}_i(\tau^1\to\tau^2)}$$

Since the side condition gives us that, $\Delta_i(\tau^1\to\tau^2)=\tau^1\to\tau^2$, we know from the definition of Δ_i that $\bar{\Delta}_i(\tau^1)=\Delta_i(\tau^1)=\tau^1$ and similarly for τ^2. From the Type Relations Lemma and the right hand premise of the **(embed)** step, we conclude that $\{\Delta\}\vdash\tau^0\lesssim_{\ell_j:i}\tau^1$ and $\{\Delta\}\vdash\tau^3\lesssim_{\ell_j:i}\tau^2$. Now notice that the reverse of $\ell_j:i$ is i :reverse(tail(ℓ_j)):j which we shall write $\ell':j$. So by the Type Relations Lemma, $\{\Delta\}\vdash\tau^1\lesssim_{\ell':j}\tau^0$. Thus we can derive:

$$\textbf{(embed)}\ \dfrac{\textbf{(var)}\ \{\Delta\};[x_i:\tau^1]\vdash_i x_i:\tau^1 \qquad \{\Delta\}\vdash\tau^1\lesssim_{\ell':j}\tau^0}{\{\Delta\};[x_i:\tau^1]\vdash_j \llcorner x_i\lrcorner_{\ell'}^{\tau^0}:\bar{\Delta}_j(\tau^0)}$$

In fact, $\{\Delta\};[x_i:\tau^1]\vdash_j \llcorner x_i\lrcorner_{\ell'}^{\tau^0}:\tau^0$ since the original derivation above provides $\Delta_j(\tau^0)=\tau^0$. It also provides $\{\Delta\};[x_j:\tau^0]\vdash_j e_j:\tau^3$. Since x_i is fresh, we may conclude $\{\Delta\};[x_i:\tau^1][x_j:\tau^0]\vdash_j e_j:\tau^3$. So by the substitution lemma, $\{\Delta\};[x_i:\tau^1]\vdash_j \{\llcorner x_i\lrcorner_{\ell'}^{\tau^0}/x_j\}e_j:\tau_3$. Thus we can derive:

$$\textbf{(abs)}\ \dfrac{\textbf{(embed)}\ \dfrac{\{\Delta\};[x_i:\tau^1]\vdash_j \{\llcorner x_i\lrcorner_{\ell'}^{\tau^0}/x_j\}e_j:\tau_3 \quad \{\Delta\}\vdash\tau^3\lesssim_{\ell_j:i}\tau^2}{\{\Delta\};[x_i:\tau^1]\vdash_i \llcorner\{\llcorner x_i\lrcorner_{\ell'}^{\tau^0}/x_j\}e_j\lrcorner_{\ell_j}^{\tau^2}:\bar{\Delta}_i(\tau^2)} \qquad \Delta_i(\tau^1)=\tau^1}{\{\Delta\};\emptyset\vdash_i \lambda x_i{:}\tau^1.\,\llcorner\{\llcorner x_i\lrcorner_{\ell'}^{\tau^0}/x_j\}e_j\lrcorner_{\ell_j}^{\tau^2}:\tau^1\to\bar{\Delta}_i(\tau^2)}$$

Since $\bar{\Delta}_i(\tau^2)=\tau^2$, the conclusion is the desired result.

$\square$

Lemma 3.12 (Progress) *For all i, if $\{\Delta\};\emptyset\vdash_i e_i:\tau$ then either e_i is some value, v_i, or there exists an e_i' such that $e_i \overset{i}{\longmapsto} e_i'$.*

Proof: By induction on the structure of e_i. There are several cases:

$e_i=x_i$ This case is impossible since $\{\Delta\};\emptyset\nvdash_i x_i:\tau$ for any τ.

$e_i=c$ Since c is a value, we are done.

$e_i=\lambda x_i{:}\tau^0.\,e_i'$ Again, e_i is a value, so the lemma holds.

$e_i=\texttt{fix}\ f_i(x_i{:}\tau^0).\,e_i'$ Rule **(4)** applies, so we can take a step.

$e_i=e_i^0e_i^2$ By the inductive hypothesis, e_i^0 is either a value or there exists some e_i^1 such that $e_i^0\overset{i}{\longmapsto}e_i^1$. If the latter holds, then by operational semantics rule **(1)** we have $e_i\overset{i}{\longmapsto}e_i^1\,e_i^2$. Otherwise, e_i^0 is a value, v_i, and another application of the inductive hypothesis gives us that e_i^2 is either a value or there exists e_i^1 such that $e_i^2\overset{i}{\longmapsto}e_i^1$. In the second case, rule **(2)** applies so e_i takes a step.

The remaining case is when $e_i^0=v_i$ and $e_i^2=v_i'$ for some values v_i and v_i'. Because e_i is well typed, Canonical Forms guarantees that $v_i=\lambda x_i{:}\tau'.\,e_i'$ for some e_i'. Thus rule **(5)** of the operational semantics applies, and e_i takes a step.

$e_i=\llcorner e_j\lrcorner_{\ell_j}^{\tau^1}$ By the well typing of e_i, the last rule used in the typing derivation must have been (**embed**). Thus we have $\{\Delta\};\emptyset\vdash_j e_j:\tau^0$ and $\{\Delta\}\vdash\tau^0\lesssim_{\ell_j:i}\tau^1$. Then by the inductive hypothesis, e_j is either a value or there is some e_j' such that $e_j\overset{i}{\longmapsto}e_j'$. In the latter case, rule **(3)** of the operational semantics applies, and we conclude that e_i takes a step.

The other possibility is that $e_j=v_j$ for some value v_j. By the Canonical Forms lemma, there are three cases based on τ^0.

$$
\begin{array}{rcl}
\underline{x_i} & = & x \\
\underline{c} & = & c \\
\underline{\lambda x_i{:}\tau.\, e_i} & = & \lambda x{:}\overline{\Phi}(\tau).\, \underline{e_i} \\
\underline{\texttt{fix } f_i(x_i{:}\tau).\, e_i} & = & \texttt{fix } f(x{:}\overline{\overline{\Phi}}(\tau)).\, \underline{e_i} \\
\underline{e_i^1\ e_i^2} & = & \underline{e_i^1}\ \underline{e_i^2} \\
\underline{\llcorner e_j \lrcorner_{\ell_j}^{\tau}} & = & \underline{e_j}
\end{array}
$$

Figure 9: Definition of Erasure $\underline{e_i}$

$\tau^0 = t$ Then $v_j = \llcorner \hat{v}_k \lrcorner_{\ell_k}^{t}$ and $t \notin \Delta_j$ for some $\hat{v}_k$ and ℓ_k, so by rule **(8)**, e_i takes a step to $\llcorner \hat{v}_k \lrcorner_{\ell_k:\ell_j}^{\tau^1}$.

$\tau^0 = \mathsf{b}$ Then $v_j = c$. If $\tau^1 = \mathsf{b}$ then $e_i \overset{i}{\longmapsto} c$ by rule **(6)** of the operational semantics. The only other possibility is that $\tau^1 = t$ for some type variable t (otherwise e_i wouldn't typecheck). If $t \in \Delta_i$ then rule **(7)** applies and $e_i \overset{i}{\longmapsto} \llcorner \hat{v}_j \lrcorner_{\ell_j}^{\bar{\Delta}_i(t)}$, otherwise e_i is the value $\llcorner c \lrcorner_{\ell_j}^{t}$.

$\tau^0 = \tau^2 \to \tau^3$ Then $v_j = \lambda x_j{:}\tau^2.\, e_j'$. If $\Delta_i(\tau^1) \neq \tau^1$, then rule **(7)** applies. Else if $\tau^1 = t$ then e_i is a value. Else τ^1 is a most-concrete arrow type, so rule **(9)** applies.

$\square$

Definition 3.13 (Stuck Term) *A term e_i is* stuck *if there does not exist a term e_i' such that $e_i \overset{i}{\longmapsto} e_i'$.*

Theorem 3.14 (Type Safety) *If $\{\Delta\}; \emptyset \vdash_i e_i : \tau$ then there is no stuck e_i' such that $e_i \overset{i}{\longmapsto}{}^* e_i'$.*

Proof: Follows directly from the Preservation and Progress Lemmas. $\square$

3.4.3 Erasure

We use embeddings to prove safety properties such as those in the next section, but the cost seems to be dynamic rules **(3)**, **(6)**, **(7)**, **(8)**, and **(9)**. Worse yet, in a language with recursion, agent lists might grow arbitrarily large. This section essentially shows that embeddings are only a proof technique. If a program evaluates to a constant, we can simply erase all embeddings and the program will still evaluate to the same constant.

More formally, we establish equivalence with the simply-typed λ-calculus by showing that embedding-erasure commutes with evaluation and constants correspond to constants. The target language has no embeddings and its semantics includes exactly those rules of the source language which do not explicitly mention embeddings, namely **(1)**, **(2)**, **(4)**, and **(5)**. Let $\underline{e_i}$ mean the erasure of e_i. Let $\overset{-}{\longmapsto}$ mean one step in the target language. Assume that if x_i and y_j are variables in a term then $x \neq y$ (α-convert as necessary). The definition of erasure is given in Figure 9. Notice that the definition depends on the implicit omniscient map Φ; we must know all type information to erase properly. If we erased to an untyped language, this would not be necessary.

Lemma 3.15 (Source and Target Evaluation Correspondence) *If $e_i \overset{i}{\longmapsto} e_i'$ then either $\underline{e_i} = \underline{e_i'}$ or $\underline{e_i} \overset{-}{\longmapsto} \underline{e_i'}$.*

Proof: By induction on the source derivation. Proceeding by cases on the last rule of the derivation:

(1) Then $\underline{e_i} = \underline{e_i^0}\ \underline{e_i^2}$ and $e_i^0 \overset{i}{\longmapsto} e_i^1$. By induction, either $\underline{e_i^0} = \underline{e_i^1}$ (in which case $\underline{e_i'} = \underline{e_i^1}\ \underline{e_i^2} = \underline{e_i}$) or $\underline{e_i^0} \overset{-}{\longmapsto} \underline{e_i^1}$ (in which case $\underline{e_i} \overset{-}{\longmapsto} \underline{e_i^1}\ \underline{e_i^2} = \underline{e_i'}$).

(2) Similar to argument for **(1)**.

(3) Then $e_i = \lceil e_j \rceil^{\tau}_{\ell_j}$ and $e_j \overset{j}{\longmapsto} e'_j$. By induction, either $\underline{e_j} = \underline{e'_j}$ (in which case $\underline{e_i} = \underline{e_j} = \underline{e'_j} = \underline{\lceil e'_j \rceil^{\tau}_{\ell_j}} = \underline{e'_i}$) or $\underline{e_j} \overset{-}{\longmapsto} \underline{e'_j}$ (in which case $\underline{e_i} = \underline{e_j} \overset{-}{\longmapsto} \underline{e'_j} = \underline{\lceil e'_j \rceil^{\tau}_{\ell_j}} = \underline{e'_i}$).

(4) We use the following substitution lemma:

Lemma 3.16 *Substitution commutes with erasure,* i.e.$\underline{\{v_i/x_i\}e_i} = \{\underline{v_i}/\underline{x_i}\}\underline{e_i}$.

Proof: By structural induction on e_i:

x_i Immediately reduces to $\underline{v_i} = \underline{v_i}$.

y_i Immediately reduces to $y = y$.

$e^1_i\ e^2_i$ Immediate from the inductive hypothesis since both substitution and erasure distribute across application.

$\lceil e_j \rceil^{\tau}_{\ell_j}$ Immediate from the inductive hypothesis since both substitution and erasure cross agent boundaries.

$\lambda y_i{:}\tau.\, e'_i$ If $x_i = y_i$, then substitution has no effect and the result is immediate. Else by the inductive hypothesis, $\underline{\{v_i/x_i\}e'_i} = \{\underline{v_i}/\underline{x_i}\}\underline{e'_i}$. So $\lambda y{:}\overline{\Phi}(\tau).\, \underline{\{v_i/x_i\}e'_i} = \lambda y{:}\overline{\Phi}(\tau).\, \{\underline{v_i}/\underline{x_i}\}\underline{e'_i}$. Finally, since $x_i \neq y_i$, this means $\underline{\{v_i/x_i\}\lambda y_i{:}\tau.\, e'_i} = \{\underline{v_i}/\underline{x_i}\}\underline{\lambda y_i{:}\tau.\, e'_i}$.

`fix` $f_i(y_i{:}\tau).\, e'_i$ Argument is completely analogous to previous case.

□

The result is immediate since $\underline{e'_i} = \underline{\{v_i/x_i\}e''_i}$ and $\underline{e_i} \overset{-}{\longmapsto} \{\underline{v_i}/x\}\underline{e''_i}$ where e''_i is the body of the source function.

(5) Follows from the substitution lemma proved in the previous case.

(6),**(7)**,**(8)** In all three cases, it is immediate from the definition of erasure that $\underline{e_i} = \underline{e'_i}$.

(9) Here $e_i = \lceil \lambda x_j{:}\tau^0.\, e_j \rceil^{\tau^1 \to \tau^2}_{\ell_j}$ and $e'_i = \lambda x_i{:}\tau^1.\, \lceil \{\lceil x_i \rceil^{\tau^0}_{l'}/x_j\}e_j \rceil^{\tau^2}_{\ell_j}$. By the definition of erasure, $\underline{e_i} = \lambda x{:}\overline{\Phi}(\tau^0).\, \underline{e_j}$ and $\underline{e'_i} = \lambda x'{:}\overline{\Phi}(\tau^1).\, \underline{\lceil \{\lceil x_i \rceil^{\tau^0}_{l'}/x_j\}e_j \rceil^{\tau^2}_{\ell_j}}$. By the substitution lemma proved in case **(4)**, and the definition of erasure, $\underline{e'_i} = \lambda x'{:}\overline{\Phi}(\tau^1).\, \{x'/x\}\underline{e_j}$. That is, $\underline{e_i}$ is α-equivalent to $\underline{e'_i}$ so long as $\overline{\Phi}(\tau^0) = \overline{\Phi}(\tau^1)$.
We prove this type equivalence using the machinery of the previous section. Augment $G(\{\Delta\})$ with edges for Φ, an omniscient agent. By the definition of Φ, if $(\tau, \tau')_i$ is an edge, then there is now a $(\tau, \tau')_{\Phi}$ edge. Since we assume e_i is well-formed, the static semantics guarantees $\tau^0 \lesssim_{\ell} \tau^1$ for some ℓ (using the Type Relations Lemma to relate parts of related arrow types). So by Relations are Paths, there is some path from τ^0 to τ^1. So there is a simple Φ path. So $\tau^0 \lesssim_{\Phi} \tau^1$. This can only be derived by **(eq)**, the premise of which is the desired result: $\overline{\Phi}(\tau^0) = \overline{\Phi}(\tau^1)$.

□

Lemma 3.17 (Type Refinement Terminates) *If* $e_i = \lceil e_j \rceil^{\tau}_{\ell_j}$ *and evaluation of* e_j *terminates (there exists* v_j *such that* $e_j \overset{j}{\longmapsto}^* v_j$*), then evaluation of* e_i *terminates.*

Proof: Consider the evaluation of e_i. By the dynamic semantics, rule **(3)** will first be applied until we have the term $\lceil v_j \rceil^{\tau}_{\ell_j}$. At this point, by Progress, one of rules **(6)**, **(7)**, **(8)**, or **(9)** must apply. For rules **(6)** or **(9)**, we reach a value in one step. For rule **(8)**, in one step one of rules **(7)**, **(8)**, or **(9)** will apply. For rule **(7)**, in one step one of rules **(8)**, or **(9)** will apply. Rule **(7)** applies at most once because $\bar{\Delta}_i(\tau) = \Delta_i(\bar{\Delta}_i(\tau))$. Finally rule **(8)** cannot apply forever because it makes a smaller inside term. □

Lemma 3.18 (Correspondence of Divergence) *If e_i diverges (has an infinite evaluation sequence), then $\underline{e_i}$ diverges.*

Proof: We prove the contrapositive: if evaluation of $\underline{e_i}$ terminates, then evaluation of e_i terminates. By Lemma 3.15 and induction on the number of evaluation steps, if $\underline{e_i} \overset{-}{\longmapsto}^* v$ then $e_i \overset{i}{\longmapsto}^* e_i'$ such that $\underline{e_i'} = v$. Now by definition of erasure, e_i' must be a v_j enclosed in some finite number of embeddings. (This is because if e_i contains an application or fix expression, it cannot erase to a value.) Now by induction on the number of embeddings, using Lemma 3.17 at each step, we conclude that evaluation of e_i' terminates. Therefore, evaluation of e_i terminates. $\square$

Theorem 3.19 *If $e_i \overset{i}{\longmapsto}^* v_i$, then $\underline{e_i} \overset{-}{\longmapsto}^* \underline{v_i}$. If e_i diverges (has an infinite length evaluation sequence), then $\underline{e_i}$ diverges.*

Proof: The first part follows from Lemma 3.15 and induction on the number of source evaluation steps. The second part is Lemma 3.18. $\square$

Corollary 3.20 *If $\{\Delta\};\emptyset \vdash_i e_i : \mathtt{b}$ and $e_i \overset{i}{\longmapsto}^* v_i$, then $\underline{e_i} \overset{-}{\longmapsto}^* v_i$.*

Proof: By type soundness, canonical forms, the preceding theorem, and the definition of erasure. $\square$

3.5 Safety Theorems

Definition 3.21 (Agents(e_i)) *Let* Agents(e_i) *be the agent names that appear anywhere in e_i (including i itself).*

Definition 3.22 (j-free) *A term e_i is said to be* j-free *if $j \notin$* Agents(e_i).

Definition 3.23 (Nearly j-free) *A term e_i is said to be* nearly j-free *if the only subterms of e_i labeled with agent j are variables.*

Note that if follows from the definition that if e_i is nearly j-free, and it does contain j-variables, then those variables are free. In particular, in order for e_i to typecheck, all j-variables occurring in e_i must be bound in the context.

Definition 3.24 (Oblivious to a Type) *A set of agents A is* <u>oblivious</u> *to t if $\forall i \in A.\ \Delta_i(t) = t$ and $\forall i \in A.\ \forall s \neq t.\ \Delta_i(s) \neq t$.*

Theorem 3.25 (Value Abstraction) *Let j be any agent. For agent $i \neq j$, let $\varphi(e_i)$ mean:*

1 $\{\Delta\}; [x_j : \bar{\Delta}_j(\mathtt{t})] \vdash_i e_i : \tau$ *for some τ*

2 e_i *is nearly j-free*

3 Agents(e_i) $\setminus \{j\}$ *are oblivious to* $\mathtt{t}$

4 If $\llcorner x_j \lrcorner^{\tau}_{\ell_j}$ is a subterm of e_i then $\tau = \mathtt{t}$.

Let $i \neq j$ be any agent. Let e_i be a term such that $\varphi(e_i)$, and let $\hat{v}_j$ and $\hat{v}_j'$ be j-primvals such that $\{\Delta\};\emptyset \vdash_i \llcorner \hat{v}_j \lrcorner^{\mathtt{t}}_j : \mathtt{t}$ and $\{\Delta\};\emptyset \vdash_i \llcorner \hat{v}_j' \lrcorner^{\mathtt{t}}_j : \mathtt{t}$. Then

$$\{\hat{v}_j/x_j\}e_i \overset{i}{\longmapsto} e_i^1 = \{\hat{v}_j/x_j\}e_i'$$

iff

$$\{\hat{v}_j'/x_j\}e_i \overset{i}{\longmapsto} e_i^2 = \{\hat{v}_j'/x_j\}e_i'$$

Furthermore, $\varphi(e_i')$.

Proof: First note that by the static semantics and the concreteness of types lemma, it must be the case that $\{\Delta\};\emptyset \vdash_j \hat{v}_j : \bar{\Delta}_j(\mathtt{t})$ and $\{\Delta\};\emptyset \vdash_j \hat{v}'_j : \bar{\Delta}_j(\mathtt{t})$.

Suppose $\{\hat{v}_j/x_j\}e_i \stackrel{i}{\longmapsto} e^1_i = \{\hat{v}_j/x_j\}e'_i$. By the soundness of the operational semantics, and the observation above, it follows that part 1 of property φ holds for e'_i.

Let σ be the substitution $\{\hat{v}_j/x_j\}$ and σ' be the substitution $\{\hat{v}'_j/x_j\}$. Note that since e_i is nearly j-free, every subterm of e_i is also nearly j-free. Similarly, since Agents$(e_i) \setminus \{j\}$ is oblivious to $\mathtt{t}$, for any subterm e''_i of e_i Agents$(e''_i) \setminus \{j\}$ is also oblivious to $\mathtt{t}$. Similarly, property 4 holds for all subterms of e_i. We use these facts in the inductive cases below.

We now show by induction on the structure of e_i that $\sigma' e_i \stackrel{i}{\longmapsto} e^2_i = \sigma' e'_i$ and $\varphi(e_i)$. (The other direction of the implication is exactly symmetric.)

$e_i = c$ This case is impossible, since there is no transition step.

$e_i = \lambda y_i{:}\tau'.\, e''_i$ This case is also impossible.

$e_i = \mathtt{fix}\ f_i(y_i{:}\tau').\, e''_i$ Since e_i is nearly j-free, $i \neq j$ and it follows that $x_j \neq f_i$ and $x_j \neq y_i$, so the definition of substitution yields $\sigma e_i = \sigma(\mathtt{fix}\ f_i(y_i{:}\tau').\, e''_i) = \mathtt{fix}\ f_i(y_i{:}\tau').\, \sigma e''_i$. The only transition rule applicable is rule **(5)**, which yields $\mathtt{fix}\ f_i(y_i{:}\tau').\, \sigma e''_i \stackrel{i}{\longmapsto} \lambda y_i{:}\tau'.\, \{\mathtt{fix}\ f_i(y_i{:}\tau').\, \sigma e''_i/f_i\}\sigma e''_i$. We can pull the substitution out front (again since the variables aren't equal) to obtain $\sigma(\lambda y_i{:}\tau'.\, \{\mathtt{fix}\ f_i(y_i{:}\tau').\, e''_i/f_i\}e''_i)$. So taking $e'_i = \lambda y_i{:}\tau'.\, \{\mathtt{fix}\ f_i(y_i{:}\tau').\, e''i/f_i\}e''_i$, we obtain $\sigma e_i \stackrel{i}{\longmapsto} \sigma e'_i$. A similar derivation using σ' in place of σ above also yields $\sigma' e_i \stackrel{i}{\longmapsto} \sigma' e'_i$.

Notice that Agents(e'_i) = Agents(e_i), so the oblivious condition still holds. Similarly, since e''_i is nearly j-free, so is e'_i. Since the subterms of e''_i satisfy item 4 of φ, so must e'_i.

$e_i = e^a_i\ e^b_i$ There are three possible transitions steps taken by terms of this form, corresponding to rules **(1)**, **(2)**, and **(4)**. Note that it follows immediately from **(app)** that $\{\Delta\}; [x_j : \bar{\Delta}_j(\mathtt{t})] \vdash_i e^a_i : \tau'' \to \tau$ and $\{\Delta\}; [x_j : \bar{\Delta}(\mathtt{t})] \vdash_i e^b_i : \tau''$ for some τ''. We consider each case separately.

(1) $\sigma e_i = \sigma(e^a_i\ e^b_i) = \sigma e^a_i\ \sigma e^b_i$. Similarly, $\sigma' e_i = \sigma'(e^a_i\ e^b_i) = \sigma' e^a_i\ \sigma' e^b_i$. Since **(1)** was used, we have $\sigma e^a_i \stackrel{i}{\longmapsto} e^{a'}_i$. The inductive hypothesis applies, so $e^{a'}_i = \sigma e^A_i$ and $\sigma' e^a_i \stackrel{i}{\longmapsto} \sigma' e^A_i$. Using rule **(1)** again, we have $\sigma e_i \stackrel{i}{\longmapsto} \sigma e^A_i\ \sigma e^b_i$,but $\sigma e^A_i\ \sigma e^b_i = \sigma(e^A_i\ e^b_i)$ by the definition of substitution. Similarly, $\sigma' e_i \stackrel{i}{\longmapsto} \sigma'(e^A_i\ e^b_i)$, so the first result holds. Furthermore, by induction, we have $\varphi(e^A_i)$, it was already observed that $\varphi(e^b_i)$, so we conclude $\varphi(e'_i)$ also holds. Likewise, Agents$(e'_i) \setminus \{j\}$ are oblivious to $\mathtt{t}$.

(2) This follows analogously to the previous subcase.

(4) Then $e_i = \lambda y_i{:}\tau'.\, e''_i\ v_i$ and, as before, $i \neq j$ so we have $y_i \neq x_j$. Calculating, we have $\sigma e_i = \sigma(\lambda y_i{:}\tau'.\, e''_i\ v_i) = \lambda y_i{:}\tau'.\, \sigma e''_i\ (\sigma v_i) = \lambda y_i{:}\tau'.\, (\sigma e''_i)\ v_i$ which transitions via rule **(4)** to $\{v_i/y_i\}(\sigma e''_i) = \sigma(\{v_i/y_i\}e''_i)$. Thus we have $\sigma e_i \stackrel{i}{\longmapsto} \sigma(\{v_i/y_i\}e''_i)$. A similar calculation using σ' in place of σ shows that $\sigma' e_i \stackrel{i}{\longmapsto} \sigma'(\{v_i/y_i\}e''_i)$, which is the desired result. Taking $e'_i = \{v_i/y_i\}e''_i$, conditions 2 through 4 hold since v_i and e''_i are subterms of e_i.

$e_i = \llcorner e_k \lrcorner^{\tau'}_{\ell_k}$ In this case, the transition rule applied must be one of **(3)**, **(6)** through **(9)**. In case **(6)** the expression is closed, so substitution has no effect and the result holds trivially. We consider each of the remaining cases separately.

(3) In this case, $k \neq j$ since the only occurrences of j subterms in e_i must be the variable x_j, which has been replaced by a value. From the definition of substitution, $\sigma e_i = \sigma \llcorner e_k \lrcorner^{\tau'}_{\ell_k} = \llcorner \sigma e_k \lrcorner^{\tau'}_{\ell_k}$. Since σe_k takes a step to $e^1_k = \sigma e'_k$, the inductive hypothesis applies and we conclude that $\sigma' e_k \stackrel{k}{\longmapsto} \sigma' e'_k$. Applying rule **(3)**, we obtain $\sigma' \llcorner e_k \lrcorner^{\tau'}_{\ell_k} = \llcorner \sigma' e_k \lrcorner^{\tau'}_{\ell_k} \stackrel{i}{\longmapsto} \llcorner \sigma' e'_k \lrcorner^{\tau'}_{\ell_k} = \sigma' \llcorner e'_k \lrcorner^{\tau'}_{\ell_k}$. By induction, $\varphi(e'_k)$ holds, so it follows immediately that $\varphi(e'_i) = \varphi(\llcorner e'_k \lrcorner^{\tau'}_{\ell_k})$ holds too.

(7) Calculating, we have: $\sigma \llcorner e_k \lrcorner^{\tau'}_{\ell_k} = \llcorner \sigma e_k \lrcorner^{\tau'}_{\ell_k} = \llcorner \hat{v}_k \lrcorner^{\tau'}_{\ell_k}$ which steps via rule **(7)** to $\llcorner \hat{v}_k \lrcorner^{\bar{\Delta}_i(\tau')}_{\ell_k} = \llcorner \sigma e_k \lrcorner^{\bar{\Delta}_i(\tau')}_{\ell_k} = \sigma \llcorner e_k \lrcorner^{\bar{\Delta}_i(\tau')}_{\ell_k}$. Thus $\tau' \neq \bar{\Delta}_i(\tau')$, and hence $\tau' \neq \mathtt{t}$. By assumption, this implies that

$k \neq j$, so property 4 is preserved. Note that since σe_k is a value, so is $\sigma' e_k$. This allows us to calculate similarly that $\sigma'\lceil e_k\rceil^{\tau'}_{\ell_k} \overset{i}{\longmapsto} \sigma'\lceil e_k\rceil^{\bar{\Delta}_i(\tau')}_{\ell_k}$. Since near j-freeness and oblivious to $\mathtt{t}$ don't depend on the typing annotation we're done.

(8) Since $\hat{v}_j$ is a j-primval, e_i must be of the form $\lceil\lceil e_h\rceil^{\mathtt{u}}_{\ell_h}\rceil^{\tau}_{\ell_k}$ such that σe_h is a h-primval. It follows that $\sigma' e_h$ is also a h-primval. We use the fact that substitutions commute with embeddings to calculate: $\sigma\lceil\lceil e_h\rceil^{\mathtt{u}}_{\ell_h}\rceil^{\tau}_{\ell_k} = \lceil\sigma\lceil e_h\rceil^{\mathtt{u}}_{\ell_h}\rceil^{\tau}_{\ell_k} = \lceil\lceil\sigma e_h\rceil^{\mathtt{u}}_{\ell_h}\rceil^{\tau}_{\ell_k}$ which transitions to $\lceil\sigma e_h\rceil^{\tau}_{\ell_h:\ell_k} = \sigma\lceil e_h\rceil^{\tau}_{\ell_h:\ell_k}$. Under the assumption that $\mathtt{u} \notin Dom(\Delta_k)$. We can then do the similar calculation for σ' to obtain $\sigma'\lceil\lceil e_h\rceil^{\mathtt{u}}_{\ell_h}\rceil^{\tau}_{\ell_k} \overset{i}{\longmapsto} \sigma'\lceil e_h\rceil^{\tau}_{\ell_h:\ell_k}$. As above, since near j-freeness and oblivious to $\mathtt{t}$ don't depend on the typing annotation on embeddings, conditions 2 and 3 hold. To show condition 4, note that k can't equal j, and so it remains to check two cases: $h = j$ and $h \neq j$.

In the first case, by assumption, $\mathtt{u} = \mathtt{t}$ and since $k \neq j$ it follows that $\tau = \mathtt{t}$, so property 4 is preserved across the step.

In the second case, $h \neq j$ and the fact that e_h is a primval, imply that e_i contains no embeddings labeled by ℓ_j and neither does e'_i. Property 4 holds vacuously.

(9) In this case, $\tau' = \tau^1 \to \tau^2$ and $e_k = \lambda x_k{:}\tau^0.\, e'_k$. Since $\tau^1 \to \tau^2 \neq \mathtt{t}$ it must be the case that $j \neq k$ and hence $x_j \neq x_k$. We calculate: $\sigma\lceil\lambda x_k{:}\tau^0.\, e'_k\rceil^{\tau^1\to\tau^2}_{\ell_k} = \lceil\sigma(\lambda x_k{:}\tau^0.\, e'_k)\rceil^{\tau^1\to\tau^2}_{\ell_k} = \lceil\lambda x_k{:}\tau^0.\, \sigma e'_k\rceil^{\tau^1\to\tau^2}_{\ell_k}$ which steps via **(9)** to

$$\lambda x_i{:}\tau^1.\, \lceil\{\lceil x_i\rceil^{\tau^0}_{\ell'}/x_k\}\sigma e'_k\rceil^{\tau^2}_{\ell_k} = \lambda x_i{:}\tau^1.\, \sigma\lceil\{\lceil x_i\rceil^{\tau^0}_{\ell'}/x_k\}e'_k\rceil^{\tau^2}_{\ell_k} = \sigma(\lambda x_i{:}\tau^1.\, \lceil\{\lceil x_i\rceil^{\tau^0}_{\ell'}/x_k\}e'_k\rceil^{\tau^2}_{\ell_k})$$

A similar calculation shows that $\sigma'\lceil\lambda x_k{:}\tau^0.\, e'_k\rceil^{\tau^1\to\tau^2}_{\ell_k} \overset{i}{\longmapsto} \sigma'(\lambda x_i{:}\tau^1.\, \lceil\{\lceil x_i\rceil^{\tau^0}_{\ell'}/x_k\}e'_k\rceil^{\tau^2}_{\ell_k})$. Since $i \neq j$, near j-freeness is preserved. The agents of $e'_i = \lambda x_i{:}\tau^1.\, \lceil\{\lceil x_i\rceil^{\tau^0}_{\ell'}/x_k\}e'_k\rceil^{\tau^2}_{\ell_k}$ are the same as for e_i, so they are oblivious as well. The embedding subterms of e'_i are either subterms of e_i, in which case they obey property 4, or labeled by either k or i, neither of which equal j, thus property 4 of φ holds for e'_i, and we're done.

□

Corollary 3.26 (Independence of Evaluation) *If*

1 $\{\Delta\}; \emptyset \vdash_i \lambda x_i{:}\mathtt{t}.\, e_i : \mathtt{t} \to b$

2 $\{\Delta\}; \emptyset \vdash_i \lceil\hat{v}_j\rceil^{\mathtt{t}}_j : \mathtt{t}$

3 $\{\Delta\}; \emptyset \vdash_i \lceil\hat{v}'_j\rceil^{\mathtt{t}}_j : \mathtt{t}$

and e_i is j-free and Agents(e_i) *are oblivious to* $\mathtt{t}$ *then:* $(\lambda x_i{:}\mathtt{t}.\, e_i)\ \lceil\hat{v}_j\rceil^{\mathtt{t}}_j \longmapsto^* c$ *iff* $(\lambda x_i{:}\mathtt{t}.\, e_i)\ \lceil\hat{v}'_j\rceil^{\mathtt{t}}_j \longmapsto^* c$

Proof (sketch): By induction on the length of the evaluation sequence using the Value Abstraction theorem.

□

Definition 3.27 (h-terms)

- *Let e_j be an <u>h-term</u> if $j = h$ or $e_j = \lceil e_k\rceil^{\tau}_{\ell_k}$ and $h \in \ell_k$.*
- *Let e_j be a <u>non-h-term</u> if $j \neq h$ or $e_j = \lceil e_k\rceil^{\tau}_{\ell_k}$ and $\exists i.\ i \in \ell_k$ and $i \neq h$.*
- *Let e_j be a <u>both-term</u> if it is an h-term and a non-h-term.*
- *Let e_j be an <u>h-embedding</u> if it is an h-term embedded in a non-h-term.*
- *Let e_j be a <u>non-h-embedding</u> if it is a non-h-term embedded in an h-term.*
- *For $\lceil e_k\rceil^{\tau}_{\ell_k}$ let τ be the <u>outside type</u> and let the type of e_k in some static derivation be the <u>inside type</u>.*

Theorem 3.28 (Host Provided Values) *Let h be a distinguished agent, the host, and let $i \neq h$ be any other agent. Let e^0_i be an i-term. Let τ_i be any base type (the input type, provided by agent i) and let τ_h be any other type (the implementation type, known only to the host). Assume the following:*

- $\Delta_h(t) = \tau_h = \Delta_h(\tau_h)$.
- $\text{Agents}(e_i^0) \setminus \{h\}$ *are oblivious to* $\mathtt{t}$ *and for all* $\mathtt{s} \neq \mathtt{t}$, $\Delta_h(s) \neq t$.
- $\{\Delta\}; \emptyset \vdash_i \lambda p_i{:}\tau_i \to \mathtt{t}.\, e_i^0 : \tau$
- *For all subterms e' of e_i^0, if e' is an h-embedding or a both-term, then t does not occur positively in its outside type; if e' is a non-h-embedding or a both-term, then t does not occur negatively in its inside type.*
- *$f_h = \lambda x_h{:}\tau_i.\, e_h$ be a pure h term such that $\{\Delta\}; \emptyset \vdash_h f_h : \tau_i \to \tau_h$. It is the h-value implementing the function p_i provided to i. Note that $\{\Delta\}; \emptyset \vdash_i \llcorner f_h \lrcorner_h^{\tau_i \to \mathtt{t}} : \tau_i \to \mathtt{t}$. It follows by rule* **(9)** *that $\llcorner f_h \lrcorner_h^{\tau_i \to \mathtt{t}} \overset{i}{\longmapsto} \lambda x_i{:}\tau_i.\, \llcorner \{\llcorner x_i \lrcorner_i^{\tau_i}/x_h\} e_h \lrcorner_h^{\mathtt{t}}$. Let $f_i = \lambda x_i{:}\tau_i.\, \llcorner \{\llcorner x_i \lrcorner_i^{\tau_i}/x_h\} e_h \lrcorner_h^{\mathtt{t}}$ and let $e_h' = \{\llcorner x_i \lrcorner_i^{\tau_i}/x_h\} e_h$.*
- $\{f_i/p_i\} e_i^0 = e^i \overset{i}{\longmapsto}{}^* e_i'$

Then if v is a value of type $\mathtt{t}$ which is a subterm of e_i', v is of the form $\llcorner \hat{v}_h \lrcorner_\ell^{\mathtt{t}}$ for a value $\hat{v}_h$ obtained by applying f_h to a suitable argument.

The next two lemmas provide what we need to prove the theorem.

Lemma 3.29 (Preservation of Host Provided Terms) *Let $\varphi(e)$ mean that for all subterms e' of e:*

1 If e' is an h-embedding or a both-term, then one of the following is true:

(a) $\mathtt{t}$ does not occur positively in the outside type of e'.

(b) The embedding is $\llcorner e_h' \lrcorner_h^{\mathtt{t}}$.

(c) $e' = \llcorner e_h'' \lrcorner_{\ell_h}^{\mathtt{t}}$ and $\exists v_i : \tau_i.\ f_i\ v_i \overset{i}{\longmapsto}{}^* \llcorner e_h'' \lrcorner_{\ell'}^{\mathtt{t}}$.

2 If e' is a non-h-embedding or a both-term, then the typing derivation of e assigns an inside type to e' in which t does not occur negatively.

Assume the same system that is assumed for the theorem.
For all i, if $\varphi(e_i)$ and $e_i \overset{i}{\longmapsto} e_i'$, then $\varphi(e_i')$.

Proof: By induction on the dynamic derivation. Last rule is:

(1) By induction.

(2) By induction.

(3) Let $e_i = \llcorner e_k \lrcorner_{\ell_k}^{\tau} \overset{i}{\longmapsto} \llcorner e_k' \lrcorner_{\ell_k}^{\tau}$. By induction, $\varphi(e_k')$. It remains to verify the term $\llcorner e_k' \lrcorner_{\ell_k}^{\tau}$ itself. Since the inside type and outside type do not change, every case is immediate except the possibility that e_i is an h-embedding or both-term and t appears positively in τ. Since e_k is closed, $\exists v_i : \tau_i$ such that $f_i\ v_i \overset{i}{\longmapsto}{}^* \llcorner e_k \lrcorner_h^t$. (So $k = h$.) Since $e_k \overset{k}{\longmapsto} e_k'$, we conclude via rule **(3)** that $f_i\ v_i \overset{i}{\longmapsto}{}^* \llcorner e_k' \lrcorner_h^t$.

(4) Follows from the following substitution lemma: If $\varphi(e_i)$ and $\varphi(v_i)$ for some value v_i, then $\varphi(\{v_i/y_i\} e_i)$. Let $\sigma = \{v_i/y_i\}$ be the substitution. Proof by structural induction on e_i:

$e_i = c$ Substitution has no effect, so the result is immediate.

$e_i = y_i$ Then $\sigma(e_i) = v_i$, and the result holds by assumption.

$e_i = y_i' \neq y_i$ Then Substitution has no effect, trivial.

$e_i = \lambda y_i'{:}\tau.\, e_i'$ By BVC, $y_i \neq y_i'$, so $\sigma(e_i) = \sigma(\lambda y_i'{:}\tau.\, e_i') = \lambda y_i'{:}\tau.\, \sigma(e_i')$ Since $\varphi(e_i)$ implies $\varphi(e_i')$, we conclude by induction that $\varphi(\sigma(e_i'))$. The outermost term satisfies φ vacuously, since it's not an embedding, therefore we conclude $\varphi(\lambda y_i'{:}\tau.\, \sigma(e_i'))$.

$e_i = \mathtt{fix}\ f_i(y_i'{:}\tau).\, e_i'$ This follows analogously to the previous case.

$e_i = e_i^0\ e_i^1$ This follows immediately from two applications of the inductive hypothesis and the fact that the outermost term of the resulting substitution is not an embedding.

$e_i = \llcorner e_k \lrcorner_{\ell_k}^{\tau}$ Then $\sigma(e_i) = \sigma(\llcorner e_k \lrcorner_{\ell_k}^{\tau}) = \llcorner \sigma(e_k) \lrcorner_{\ell_k}^{\tau}$. Since $\varphi(e_i)$ implies $\varphi(e_k)$, it follows by induction that $\varphi(\sigma(e_k))$. Thus the only subterm that needs to be checked is $\llcorner \sigma(e_k) \lrcorner_{\ell_k}^{\tau}$ itself. Since substitution doesn't change types, conditions 1(a) and 2 follow immediately. The remaining case occurs when $\mathtt{t}$ occurs positively in τ. Since $\varphi(e_i)$, we have $\tau = \mathtt{t}$. In the case that $e_k = e_h'$, its only free variable is x_i. If $y_i = x_i$, then $\sigma(e_k) = \sigma(e_h') = \sigma(\{\llcorner x_i \lrcorner_i^{\tau_i}/x_h\}e_h) = \{\llcorner v_i \lrcorner_i^{\tau_i}/x_h\}e_h$. Notice that $f_i\ v_i$ transitions in one step by **(4)** to $\{\llcorner v_i \lrcorner_i^{\tau_i}/x_h\}e_h$, so 1(c) is satisfied. If $y_i \neq x_i$ then the substitution has no effect and 1(b) is still satisfied. The only other possibility is that e_i satisfied 1(c), in which case e_k is closed and hence substitution has no effect. Thus in all cases, we conclude $\varphi(\sigma(e_i))$.

(5) All of the subterms of e_i' are subterms of e_i with the exception of e_i itself. Since e_i' is not an embedding, the conditions hold trivially for it and we're done.

(6) Let $e_i = \llcorner \llcorner \hat{v}_h \lrcorner_{\ell_h}^{\mathtt{u}} \lrcorner_{\ell_k}^{\tau} \overset{i}{\longmapsto} \llcorner \hat{v}_h \lrcorner_{\ell_h:\ell_k}^{\tau}$ By the constraints on the system, if τ contains $\mathtt{t}$, then $\mathtt{u} = \tau = \mathtt{t}$. (Otherwise some $\Delta(\mathtt{u})$ contains $\mathtt{t}$.) The result is immediate if τ does not contain $\mathtt{t}$, so assume it does. By the system, static semantics, and canonical forms, $\hat{v}_h$ must be of type τ_i, which is devoid of $\mathtt{t}$'s. So the inside type cannot have a negative occurrence of $\mathtt{t}$. If $h \notin \ell_k$, then we're done because appending non-h agents cannot violate $\varphi(e_i')$ since the inside type has no negative occurrence of $\mathtt{t}$. To see that h cannot occur in ℓ_k, note that $e_k \neq e_h'$. Furthermore, since f_h accepts only base values, any term stepped to by $f_h\ v$ must be a pure h term. This rules out the possibility that e_i satisfies 1(c), and so the only possibility is that h does not occur in ℓ_k.

(7) Immediate because for all s and k, $\Delta_k(s) \neq t$.

(8) Immediate.

(9) Let $e_i = \llcorner \lambda x_j{:}\tau^0.\, e_j \lrcorner_{\ell_j}^{\tau^1 \to \tau^2} \overset{i}{\longmapsto} \lambda x_i{:}\tau^1.\, \llcorner \{\llcorner x_i \lrcorner_{i:\mathtt{rev}(tl(\ell_j))}^{\tau^0}/x_j\}e_j \lrcorner_{\ell_j}^{\tau^2} = e_i'$

If e_i is a both term, then $\mathtt{t}$ does not occur positively in $\tau^1 \to \tau^2$ and $\mathtt{t}$ does not occur positively in τ^0 (since the inside type must be $\tau^0 \to \tau^3$ for some τ^3). The $\mathtt{t}$ doesn't occur positively in τ^2 or negatively in τ^1. This immediately verifies e_i' and the body of the function, so long as the embedded term is verified. We know $\varphi(e_j)$. We need a substitution lemma analogous to the one above but tailored to substitutions of the form $\llcorner x_i \lrcorner_{\ell_i}^{\tau^0}$. The key observation is that the types prohibit $\{\llcorner x_i \lrcorner_{i:\mathtt{rev}(tl(\ell_j))}^{\tau^0}/x_j\}e_j$ from being e_h', or the result of $f_i\ v_i$ for any v_i. Thus it is sufficient to show $\varphi(\llcorner x_i \lrcorner_{i:\mathtt{rev}(tl(\ell_k))}^{\tau^0})$. Since $\mathtt{t}$ does not occur positively in τ^0, it suffices to show that $\mathtt{t}$ does not occur negatively in the type of x_i. That type is τ^1, so we are done.

If e_i is an h-embedding and not a both-term, then $\mathtt{t}$ does not occur positively in $\tau^1 \to \tau^2$ (see that the other cases are not possible) and τ^0 does not contain $\mathtt{t}$ at all (since $k = h$). This is stronger than what we assumed in the previous case, so apply that argument.

If e_i is a non-h-embedding and not a both-term, then $\mathtt{t}$ does not occur at all in $\tau^1 \to \tau^2$ (by the side-condition for rule 5 which gives rule 7 precedence) and does not occur positively in τ^0. Again, this is stronger than what we assumed in the both-term case.

□

Lemma 3.30 ($\mathtt{t}$ values are h-embeddings) *Assume the system of the theorem. If v is a value of type $\mathtt{t}$ and a subterm of e_i, then v is h-term embedded in a non-h-term with a τ_i for the embedded term.*

Proof: Follows directly from the system and the static semantics. □

The theorem follows from the lemmas and claiming that if a value is produced by check, then it is permissible (applying f to it produces true).

4 Future Work

4.1 Polymorphism

Type abstraction and polymorphism are closely related. Indeed, the example encoding of an agent in the introduction used polymorphism to achieve type abstraction. However, the two are different. The key distinction seems to be one of scope. Our type abstractions are globally scoped and statically known. Polymorphism allows locally scoped type abstractions which can be instantiated many times at run-time.

There are several approaches to adding polymorphism to the multiagent calculus, none of which we have fully explored. One is to simply add polymorphism, keeping the type variables for polymorphism and agents disjoint. The necessary additions seem to be straightforward.

A different avenue is to encode polymorphism using the embeddings of the multiagent calculus. The idea is to represent the body of a polymorphic function, $\Lambda\alpha.\, e_i$ as an agent with no information about the type variable α. When such a function is applied to a type τ, a new agent, j, that knows $\alpha = \tau$ is "spawned" with the body e_i embedded inside it. The type system prevents the body of the function from breaking the type abstraction, while the "wrapper" agent, j, provides a way to recover the type information when the function returns.

4.2 Beyond Type Abstraction

Type abstraction is one application of formalizing the notion of principal. The difference between agents in this calculus is what type-information is available to them. There are many other dimensions along which this idea can be extended. We can use essentially the same mechanism to formalize foreign function calls, where each agent uses a different set of operational rules. For example, we could give some agents a call-by-name semantics, allowing a mixture of eager and lazy evaluation. For less similar languages, the embeddings express exactly where foreign data conversions and calling conventions need to occur.

The Δ_i's capture an agent's view of its environment. In this paper we restricted our attention to type information, but this too can potentially be extended. The Δ_i's could represent arbitrary capabilities, controlling access to resources in the environment. Suitable rules in the operational semantics would propagate which capabilities are available. The ability to update the Δ_i's could be reflected into the language itself, yielding a dynamic system in which a principal could grant or revoke capabilities to other agents at run-time.

5 Related Work

Perhaps the closest work to ours is Leroy and Rouaix's investigation into the safety properties of typed applets [9]. They use a λ-calculus augmented with state in order to prove theorems similar to Theorem 2.16. They too distinguish between the execution environment code and applet code, similar to our use of principals, but they consider only the two-agent case and take a less syntactic approach.

There has been much work on representation independence and parametric polymorphism, as pioneered by Strachey [18] and Reynolds [17]. Such notions have been incorporated into programming languages such as SML and Haskell [10, 14] and studied extensively in Girard's system F [5].

Abadi, Cardelli, and Curien have taken a syntactic approach to parametricity by formalizing the logical relations arguments used in such proofs [1]. More recently, Crary has proposed the use of singleton types as a means of proving parametricity results without resorting to the construction of models [4]. Pierce and Sangiorgi [15] have studied parametricity in a polymorphic variant of the π-calculus; they too are concerned with multiple views of the same value and point out some interesting interactions between abstraction and aliasing.

None of the above work (except Leroy and Rouaix's) explicitly involves the notion of principal. Our syntactic separation of agents is similar to Nielson and Nielson's Two-Level λ-calculus [13]. There they are concerned with binding time analysis, so the two principals' code is inherently not mixed during evaluation. A notion of principal also arises in the study of language based security, where privileged agents may not leak information to unprivileged ones. See, for example, Heinze and Reicke's work on the SLam calculus [8], Volpano and Smith's work on type-based security [19], or Myers' JFlow system [12].

6 Conclusion

We have created a multiagent calculus in which the notion of principal is made explicit in the language. This syntactic distinction allows us to track agent code during evaluation, giving syntactic proofs of interesting type abstraction properties. Our hope is that these techniques will scale to realistic, hard to model languages.

References

[1] Martín Abadi, Luca Cardelli, and Pierre-Louis Curien. Formal parametric polymorphism. In *Principles of Programming Languages*, volume 20, pages 157–167, January 1993.

[2] Godmar Back, Patrick Tullmann, Leigh Stoller, Wilson C. Hseih, and Jay Lepreau. Java operating systems: Design and implementation. Technical Report UUCS-98-015, University of Utah, August 1998.

[3] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers, and Craig Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995.

[4] Karl Crary. A simple proof technique for certain parametricity results. Technical Report CMU-CS-98-185, Carnegie Mellon University, December 1998.

[5] Jean-Yves Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[6] Michael Godfrey, Tobias Mayr, Praveen Seshadri, and Thorsten von Eicken. Secure and portable database extensibility. In *Proceedings of the 1997 ACM-SIGMOD Conference on the Management of Data*, pages 390–401, Seattle, WA, June 1998.

[7] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorston von Eiken. Implementing multiple protection domains in Java. In *1998 USENIX Annual Technical Conference*, New Orleans, LA, June 1998.

[8] Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Conference Record of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 365–377. ACM Press, 1998.

[9] Xavier Leroy and François Rouaix. Security properties of typed applets. In *Principles of Programming Languages*, January 1998.

[10] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[11] J.C. Mitchell. On the equivalence of data representations. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 305–330. Academic Press, 1991.

[12] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, TX, January 1999.

[13] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Number 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[14] John Peterson, Kevin Hammond, Lennart Augustsson, Brian Boutel, Warren Burton, Joseph Fasel, Andrew D. Gordon, John Hughes, Paul Hudak, Thomas Johnsson, Mark Jones, Erik Meijer, Simon Peyton Jones, Alastair Reid, and Philip Wadler. Report on the programming language Haskell, version 1.4. http://www.haskell.org/report.

[15] Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. Technical Report MS-CIS-99-10, University of Pennsylvania, April 1999. (Summary in POPL '97).

[16] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, Paris, France, April 1974.

[17] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A Mason, editor, *Information Processing*, pages 513–523. Elsevier Science Publishers B.V., 1983.

[18] C. Strachey. Fundamental concepts in programming languages. Unpublished Lecture Notes, Summer School in Computer Programming, August 1967.

[19] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Proceedings of TAPSOFT '97, Colloquium on Formal Approaches to Software Engineering*, Lille, France, April 1997.

[20] Steve Zdancewic, Dan Grossman, and Greg Morrisett. Principals in programming languages: A syntactic proof technique. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*, Paris, France, September 1999.