Principals in Programming Languages: A Syntactic Proof Technique

Principals in Programming Languages: A Syntactic Proof Technique

Steve Zdancewic Dan Grossman Greg Morrisett [*]

Department of Computer Science
Cornell University

Abstract

Programs are often structured around the idea that different pieces of code comprise distinct *principals*, each with a view of its environment. Typical examples include the modules of a large program, a host and its clients, or a collection of interactive agents.

In this paper, we formalize this notion of principal in the programming language itself. The result is a language in which intuitive statements such as, "the client must call open to obtain a file handle," can be phrased and proven formally.

We add principals to variants of the simply-typed λ-calculus and show how we can track the code corresponding to each principal throughout evaluation. This multiagent calculus yields syntactic proofs of some type abstraction properties 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.

```
(* File handle implemented as int *)
abstype fh
open : string → fh
read : fh → char
```

Figure 1: Abstract interface for file handles

Consider a host-provided interface for an abstract type of file handles, fh, and operations to create and use them (Figure 1). 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 fh = *int*, while clients do not.

The conventional wisdom is that the use of 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 cannot 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.

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 [18] (see Figure 2). Here, the type fh is held abstract by encoding the agent as a polymorphic function. We can then appeal to Reynolds' parametricity results [19] to conclude that the agent respects the host's interface.

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

Figure 2: Polymorphic λ-calculus agent encoding

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,

[*] 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.

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. As Figure 3 shows, 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.

$$\tau_h = \{\mathtt{open} : \mathit{string} \rightarrow \mathsf{fh}, \mathtt{read} : \mathsf{fh} \rightarrow \mathit{char}\}$$

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

Figure 3: "Linking" agent code

The next section describes a two-agent setting sufficient for proving interesting properties about the file handle example. It then discusses how to include more advanced language features, such as recursive types and state. Section 3 introduces a multiagent calculus 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.

$$\begin{array}{lcl}
\tau & ::= & \mathsf{t} \mid \mathsf{b} \mid \tau \rightarrow \tau' \\
\\
A & ::= & x_a \mid c \mid \lambda x_a{:}\tau.\, A \mid A\ A' \mid [H]^{\tau}_{h} \\
\hat{A} & ::= & c \mid \lambda x_a{:}\tau.\, A \mid [\hat{H}]^{\mathsf{t}}_{h} \\
\\
H & ::= & x_h \mid c \mid \lambda x_h{:}\tau.\, H \mid H\ H' \mid [A]^{\tau}_{a} \\
\hat{H} & ::= & c \mid \lambda x_h{:}\tau.\, H
\end{array}$$

Figure 4: Two-agent Syntax

Figure 4 gives the syntax for the two-agent calculus. Types, τ, include a base type, b, the host's abstract type, t, and function 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.

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 and host terms mix during evaluation. To keep track of this intermingling, agent terms contain *embedded* host terms of the form $[H]^{\tau}_{h}$. 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 t during execution. In particular, a host term of type τ_h may be embedded in an agent term. If the annotation is t, then the agent has no information about the form of the term inside the embedding. Thus, an embedding with annotation t containing a host value is an agent value.

A good intuition for the semantics is to imagine two copies of the simply-typed λ-calculus augmented with a new type t. In the agent copy, t is abstract, while in the host copy, we have $\mathsf{t} = \tau_h$. Because the host has more knowledge, there is an asymmetry in the language. In the semantics, this asymmetry manifests itself in rules in which the host refines t to τ_h.

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 well-formed 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 a polychromatic rule have the same color, and the rule is short-hand for two analogous rules, one for each color.

We write $\{e'/x\}e$ for the capture-avoiding substitution of e' for x in e. Terms are equal up to α-conversion, where substituted variables are of the same color. We also define substitution on types, written $\{\tau'/\mathsf{t}\}\tau$. Intuitively, we use the substitution $\{\tau_h/\mathsf{t}\}\tau$ to produce the host's view of τ.

We say an agent term is *host-free* if it contains no embeddings (and similarly for *agent-free* host terms).

2.3 Dynamic Semantics

Figure 5 describes a small-step operational semantics for the two-agent calculus. The polychromatic rules are the same as for the simply-typed, call-by-value λ-calculus. The other rules handle embeddings.

Rules **(A1)** and **(H1)** 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 **(A2)** and **(H2)**).

Rules **(A3)** and **(H3)** maintain the distinction between agent and host code. For example, suppose the agent contains a host function that is being exported at type $\tau' \rightarrow \tau''$.

$$
\begin{array}{rrrcll}
\text{Polychrome Steps} & (\mathbf{P1}) & e\ e' & \longmapsto & e''\ e' & \text{if } e \longmapsto e'' \\
 & (\mathbf{P2}) & \hat{e}\ e' & \longmapsto & \hat{e}\ e'' & \text{if } e' \longmapsto e'' \\
 & (\mathbf{P3}) & (\lambda x{:}\tau.\, e)\ \hat{e} & \longmapsto & \{\hat{e}/x\}e & \\
\text{Agent Steps} & (\mathbf{A1}) & [H]_h^{\tau} & \longmapsto & [H']_h^{\tau} & \text{if } H \longmapsto H' \\
 & (\mathbf{A2}) & [c]_h^{b} & \longmapsto & c & \\
 & (\mathbf{A3}) & [\lambda x_h{:}\tau.\, H]_h^{\tau' \to \tau''} & \longmapsto & \lambda x_a{:}\tau'.\, [\{[x_a]_a^{\tau}/x_h\}H]_h^{\tau''} & \\
\text{Host Steps} & (\mathbf{H1}) & [A]_a^{\tau} & \longmapsto & [A']_a^{\tau} & \text{if } A \longmapsto A' \\
 & (\mathbf{H2}) & [c]_a^{b} & \longmapsto & c & \\
 & (\mathbf{H3}) & [\lambda x_a{:}\tau.\, A]_a^{\tau' \to \tau''} & \longmapsto & \lambda x_h{:}\{\tau_h/\mathsf{t}\}\tau'.\, [\{[x_h]_h^{\tau}/x_a\}A]_a^{\tau''} & \\
 & (\mathbf{H4}) & [[\hat{H}]_h^{\mathsf{t}}]_a^{\tau_h} & \longmapsto & \hat{H} &
\end{array}
$$

Figure 5: Two-agent Dynamic Semantics

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 $\tau' = \mathsf{t}$.

Thus, **(A3)** 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, $[x_a]_a^{\tau}$. This embedding is exported to the host at type τ, the type the host originally expected for the function argument. The rule for hosts, **(H3)**, is symmetric, except that because the host may use t as τ_h, occurrences of t in the host function's type annotation are replaced by τ_h.

The final rule, **(H4)**, allows the host to "open up" an agent value that is really an embedded host value. This allows the host to recover a value that has been embedded abstractly in the agent.

The crucial point is that any attempt by the agent to treat a value of type t as a function 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 2.6.

2.4 Static Semantics

Figure 7 describes the static semantics for the two-agent calculus. The typing context, Γ, maps variables (of either color) to types. The polychromatic rules are standard, as is the introduction rule for agent functions. For host functions, the only difference is that t is not allowed to appear in the annotation for the argument to a function. Since the host knows that $\mathsf{t} = \tau_h$, this does not limit expressiveness. The convenient effect of this side condition is that types of host terms never contain t. The presence of t would complicate other rules such as **(var)** and **(app)**, as well as the statement of Preservation, because additional type refinement would be necessary.

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 t) has type τ if the host is able to show that the "actual" type of H is $\{\tau_h/\mathsf{t}\}\tau$. In other words, the host may hide type information from the agent by replacing some occurrences of τ_h with 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 Examples

We now give two examples of program evaluation in the two-agent calculus. Returning to our file handle example, let $\mathsf{t} = \mathsf{fh}$ and $\tau_h = int$.

Figure 6 shows the agent obtaining a file handle through a host interface. For simplicity, only the host's `open` function is provided to the agent. The host implementation, `ho`, takes in a string and produces an integer representing a file handle. This code is embedded inside the agent at the more abstract type $string \to \mathsf{fh}$.

$$
\begin{array}{ll}
 & (\lambda \mathtt{open}_a{:}string \to \mathsf{fh}.\, \mathtt{open}_a\ \text{``}myfile\text{''}) \\
 & \quad [\lambda \mathtt{s}_h{:}string.\, \mathtt{ho}(\mathtt{s}_h)]_h^{string \to \mathsf{fh}} \\
(1) & (\lambda \mathtt{open}_a{:}string \to \mathsf{fh}.\, \mathtt{open}_a\ \text{``}myfile''\text{)} \\
 & \quad \lambda \mathtt{s}_a{:}string.\, [\mathtt{ho}([\mathtt{s}_a]_a^{string})]_h^{\mathsf{fh}} \\
(2) & \longmapsto \lambda \mathtt{s}_a{:}string.\, [\mathtt{ho}([\mathtt{s}_a]_a^{string})]_h^{\mathsf{fh}}\ \text{``}myfile'' \\
(3) & \longmapsto [\mathtt{ho}([\text{``}myfile'']_a^{string})]_h^{\mathsf{fh}} \\
(4) & \longmapsto [\mathtt{ho}(\text{``}myfile'')]_h^{\mathsf{fh}} \\
(n) & \longmapsto^{*} [3]_h^{\mathsf{fh}}
\end{array}
$$

Figure 6: Agent calling `open`

Step (1) uses **(A3)** to convert the embedded host function to an agent function. Note that the new variable, $\mathtt{s}_a$, is

Polychrome Rules

$$(\textbf{var})\quad \Gamma \vdash x : \Gamma(x) \qquad (\textbf{const})\quad \Gamma \vdash c : b \qquad (\textbf{app})\quad \frac{\Gamma \vdash e : \tau' \to \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \tau}$$

Agent Rules

$$(\textbf{HinA})\quad \frac{\Gamma \vdash H : \{\tau_h/\mathsf{t}\}\tau}{\Gamma \vdash \llbracket H \rrbracket_h^{\tau} : \tau} \qquad (\textbf{Afn})\quad \frac{\Gamma[x_a : \tau'] \vdash A : \tau}{\Gamma \vdash \lambda x_a{:}\tau'.\, A : \tau' \to \tau}$$

Host Rules

$$(\textbf{AinH})\quad \frac{\Gamma \vdash A : \tau}{\Gamma \vdash \llbracket A \rrbracket_a^{\tau} : \{\tau_h/\mathsf{t}\}\tau} \qquad (\textbf{Hfn})\quad \frac{\Gamma[x_h : \tau'] \vdash H : \tau}{\Gamma \vdash \lambda x_h{:}\tau'.\, H : \tau' \to \tau}(\mathsf{t} \notin \tau')$$

Figure 7: Two-agent Static Semantics

embedded in the host as an agent term. Step (2) is a standard β-reduction. Step (3) is another β-reduction, passing in the string "$myfile$". Step (4) uses **(H2)** to extract the string from the embedding. At this point, the host function `ho` is applied to a host value. Repeated use of **(A1)** allows the host function to proceed. We assume that `ho` returns 3 when applied to "$myfile$". This result, embedded within the agent code at type fh, is an agent value.

$$\begin{array}{lll}
 & & \llbracket \lambda\mathtt{handle}_h{:}int.\, \mathtt{hr}(\mathtt{handle}_h) \rrbracket_h^{\mathsf{fh}\to char} \quad \llbracket 3 \rrbracket_h^{\mathsf{fh}} \\
(1) & \longmapsto & \lambda\mathtt{handle}_a{:}\mathsf{fh}.\, \llbracket \mathtt{hr}(\llbracket \mathtt{handle}_a \rrbracket_a^{int}) \rrbracket_h^{char} \quad \llbracket 3 \rrbracket_h^{\mathsf{fh}} \\
(2) & \longmapsto & \llbracket \mathtt{hr}(\llbracket \llbracket 3 \rrbracket_h^{\mathsf{fh}} \rrbracket_a^{int}) \rrbracket_h^{char} \\
(3) & \longmapsto & \llbracket \mathtt{hr}(3) \rrbracket_h^{char} \\
(n) & \longmapsto^* & \llbracket \mathtt{'A'} \rrbracket_h^{char} \\
 & \longmapsto & \mathtt{'A'}
\end{array}$$

Figure 8: Agent calling `read`

The second example (Figure 8) illustrates the agent calling the host's `read` function, passing in the file handle $\llbracket 3 \rrbracket_h^{\mathsf{fh}}$. The host code for `read` is embedded in the agent and exported at the type, fh → $char$. (We omit the linking steps for brevity.) The body of the host function is `hr`, a host term taking an integer representing a file handle and returning an integer read from that file.

In step (1), the agent extracts the host function via rule **(A3)**. The type of the argument $\mathtt{handle}_a$ is abstract in the agent, so the type annotation is changed to fh. The second step is a β-reduction. At this point, evaluation continues via **(H4)**, which in step (3) allows the embedded host code to extract the integer 3, held abstract by the agent, as a regular value. The application $\mathtt{hr}(3)$ proceeds as usual, until the host computes the character read from the file. At last, since this embedded character is exported to the agent as such, rule (**A2**) produces the value `'A'`.

2.6 Safety Properties

In this section, we explore properties of the two-agent calculus including soundness and some type abstraction theorems. Sketches of the proofs are deferred as they are corollaries to the corresponding proofs in Section 3.5. 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.

The following lemmas establish type soundness:

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

- b*, then $\hat{e} = c$ for some c.*
- *$\tau' \to \tau''$, then $\hat{e} = \lambda x{:}\tau'.\, e'$ for some x and e'.*
- t*, then $\hat{e} = \llbracket \hat{H} \rrbracket_h^{\mathsf{t}}$ for some $\hat{H}$ of type τ_h.*

Lemma 2.2 (Preservation) *If $\emptyset \vdash e : \tau$ and $e \longmapsto e'$ then $\emptyset \vdash e' : \tau$.*

Lemma 2.3 (Progress) *If $\emptyset \vdash e : \tau$, then either e is a value or there exists an e' such that $e \longmapsto e'$.*

Definition 2.4 (Stuck Terms) *A term e is stuck if it is not a value and there is no e' such that $e \longmapsto e'$.*

Theorem 2.5 (Type Soundness) *If $\emptyset \vdash e : \tau$ then there is no stuck e' such that $e \longmapsto^* e'$.*

Given a term, if we ignore the distinction between colors, erase the embeddings, and replace `t` with τ_h, we have a simply-typed λ-calculus term. Formally, Figure 9 defines the erasure of a two-agent term. (All rules are polychromatic.) The following lemma states that erasure commutes with evaluation.

Lemma 2.6 (Erasure) *Let e be any two-agent term such that $\emptyset \vdash e : \tau$. Then $e \longmapsto^* \hat{e}$ iff $erase(e) \longmapsto^* erase(\hat{e})$.*

The interesting fact is that the erasure of rule **(A3)** is basically $\lambda x{:}\tau.\, e \longmapsto \lambda x'{:}\tau.\, \{x'/x\}e$. The two terms are α-equivalent.

With soundness and erasure established, we re-examine the abstraction properties of the introduction. Since the host is always capable of providing information to the agent, we are particularly interested in evaluations where we assume the host does not do so. Toward this end, we define *host-free* to describe an agent term which has no host terms embedded in it.

One desirable property of the file handle interface is that the agent never breaks the type abstraction for file handles. For example, if $\emptyset \vdash handle : \mathsf{fh}$, $\emptyset \vdash \lambda\mathtt{f}{:}\mathsf{fh}.\, A : \tau$, and A

$$\begin{aligned}
erase(x) &= x\\
erase(c) &= c\\
erase(\lambda x{:}\tau.\, e) &= \lambda x{:}\{\tau_h/\mathtt{t}\}\tau.\, erase(e)\\
erase(e\ e') &= erase(e)\ erase(e')\\
erase([e]_i^\tau) &= erase(e)
\end{aligned}$$

Figure 9: Two-agent *erase* translation

is host-free, then $(\lambda \mathtt{f}{:}\mathsf{fh}.\, A)$ *handle* never steps to a context where *handle* is treated as an integer. This property is a corollary of type soundness.

Another property is that the agent is oblivious to the particular choice of integers used by the host to represent a given file handle. More formally:

Theorem 2.7 (Independence of Evaluation) *If $[\hat{H}]_h^{\mathsf{fh}}$ and $[\hat{H}']_h^{\mathsf{fh}}$ are well-typed, A is host-free, and $\emptyset \vdash \lambda \mathtt{f}_a{:}\mathsf{fh}.\, A : \mathsf{fh} \to b$, then:*

$$\begin{gathered}
(\lambda \mathtt{f}_a{:}\mathsf{fh}.\, A)\ [\hat{H}]_h^{\mathsf{fh}} \longmapsto^* c\\
\text{iff}\\
(\lambda \mathtt{f}_a{:}\mathsf{fh}.\, A)\ [\hat{H}']_h^{\mathsf{fh}} \longmapsto^* c
\end{gathered}$$

The proof strengthens the claim to a step-by-step evaluation correspondence when using $\hat{H}$ and $\hat{H}'$:

Lemma 2.8 (Value Abstraction) *Let $\hat{H}$ and $\hat{H}'$ be host values of type τ_h. If A is host-free, $[x_a : \mathsf{fh}] \vdash A : \tau$, and $\{[\hat{H}]_h^{\mathsf{fh}}/x_a\}A$ is not a value, then there exists a host-free term A' such that:*

- $[x_a : \mathsf{fh}] \vdash A' : \tau$
- $\{[\hat{H}]_h^{\mathsf{fh}}/x_a\}A \longmapsto \{[\hat{H}]_h^{\mathsf{fh}}/x_a\}A'$
- $\{[\hat{H}']_h^{\mathsf{fh}}/x_a\}A \longmapsto \{[\hat{H}']_h^{\mathsf{fh}}/x_a\}A'$

The embeddings also enable us to track expressions of the abstract type during evaluation, thus allowing us to formalize a third property: the agent must have called `open` to obtain a file handle.

Theorem 2.9 (File handles come from `open`) *Suppose $(\lambda \mathtt{open}_a{:}string \to \mathsf{fh}.\, A)$ is well-typed and A is host-free. If the application of $(\lambda \mathtt{open}_a{:}string \to \mathsf{fh}.\, A)$ to $[\lambda \mathtt{s}_h{:}string.\, \mathtt{ho}(\mathtt{s}_h)]_h^{string\to\mathsf{fh}}$ steps to some term A' containing $[\hat{H}]_h^{\mathsf{fh}}$ as a subterm, then $\hat{H}$ was derived from a sequence of the form $\mathtt{ho}(s) \longmapsto^* \hat{H}$.*

The proof shows that (after one step) every host embedding has as its inside term either an application of `ho` or an intermediate result of such an application.

2.7 Extending the Language

The two-agent calculus lacks many features found in realistic programming languages, but our embedding approach scales to handle many additions.

It is straightforward to add standard type constructors such as products, records, sums, *etc.* The static semantics are standard, while the dynamic semantics, in addition to the usual rules, also include means of propagating embeddings. For example, $[(\hat{A}, \hat{A}')]_a^{\tau\times\tau'} \longmapsto ([\hat{A}]_a^\tau, [\hat{A}']_a^{\tau'})$.

$$\begin{aligned}
\tau &::= \ldots \mid \alpha \mid \mu\alpha.\tau\\
H &::= \ldots \mid \mathtt{roll}_{\mu\alpha.\tau}\ H \quad \mid \mathtt{unroll}_{\mu\alpha.\tau}\ H\\
\hat{H} &::= \ldots \mid \mathtt{roll}_{\mu\alpha.\tau}\ \hat{H}
\end{aligned}$$

$$[\mathtt{roll}_{\mu\alpha.\tau}\ \hat{A}]_a^{\mu\alpha.\tau} \longmapsto \mathtt{roll}_{\mu\alpha.\{\tau_h/\mathtt{t}\}\tau}\ [\hat{A}]_a^{\{\mu\alpha.\tau/\alpha\}\tau}$$

Figure 10: Adding recursive types

Recursive types also pose no problems. The necessary additions and an example host dynamic step are shown in Figure 10. (The agent rule is similar.) Our type abstraction results carry through even in this setting.

Adding state is more subtle. A rigorous treatment of references appears in the companion technical report [25]. We have proven soundness, as well as stateful analogues of the abstraction theorems presented here.

The key issue is how to facilitate the sharing of heap values between the host and agent. Our approach assigns (possibly abstract) types to heap locations and records which agent last assigned to a location. An ensuing dereference by the other agent produces an embedded term. All of this extra information can be erased, so the target language of erasure is a standard language with references. An interesting technical issue is that it is unsound for the host to export a $\mathtt{t}$ ref value at type τ_h ref. Therefore, the definition of type substitution is modified so as not to perform substitution under the ref constructor.

Restating the type abstraction properties to account for the heap is somewhat trickier, since we must preclude the possibility that the host leaks information to the agent through a shared reference. Essentially, the notion of host-free must be extended to account for terms reachable through references.

With references and recursive types, we expect that our results could be extended to handle threads in the style of those in Reppy's doctoral thesis [17]. Such scalability is due to the similarity of our proofs to standard subject-reduction, as popularized by Wright and Felleisen [24]. Namely, a property of interest is shown to be invariant during evaluation. Although these syntactic techniques yield weaker results than semantic accounts of parametricity, they are useful in practice because we do not have to build a model containing recursive types, references, and threads.

3 The Multiagent Language

So far, we have described a simple 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, and, once that has been introduced, agents should be able to share type information.

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

Figure 11: Multiagent Syntax

Generalizing the language has another benefit: we can prove theorems once for a broad class of systems. The type abstraction properties for an instance of the system (such as our two-agent calculus) follow as corollaries.

3.1 Syntax

Figure 11 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 metavariables i and j range over $\{1, \ldots, n\}$.

The terms for agent i include variables, x_i, non-recursive functions, $\lambda x_i{:}\tau.\, e_i$, recursive functions, $\texttt{fix}\ f_i(x_i{:}\tau).e_i$, function applications, $e_i\ e_i'$, and embeddings $[e_j]^{\tau}_{\ell_j}$. We include both recursive and non-recursive functions to simplify the dynamic semantics (see rule **(4)** in Figure 12).

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 $\texttt{rev}(\ell_i)$ 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. 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 $\mathsf{fh} = string$.)

To capture this information, we assume that for each agent i there is a finite partial map from type variables to types called δ_i. Each δ_i extends naturally to a map, Δ_i, from an arbitrary type τ as follows:

$$\begin{array}{rcl}
\Delta_i(\mathsf{b}) & = & \mathsf{b} \\
\Delta_i(t) & = & \left\{ \begin{array}{ll} t & t \notin \mathrm{Dom}(\delta_i) \\ \delta_i(t) & t \in \mathrm{Dom}(\delta_i) \end{array} \right. \\
\Delta_i(\tau^1 \rightarrow \tau^2) & = & \Delta_i(\tau^1) \rightarrow \Delta_i(\tau^2)
\end{array}$$

To maintain consistency between agents, we require that for every pair of agents, i and j, if $t \in \mathrm{Dom}(\delta_i) \cap \mathrm{Dom}(\delta_j)$, then $\delta_i(t) = \delta_j(t)$.[1] We further restrict the δ_i maps so that they interact in a well-founded manner. For example, we do not allow $\delta_i(t) = t \rightarrow t$, or the more subtle $\delta_i(t) = s \rightarrow s$, $\delta_j(s) = t$. In general, we assume the collection of type variables can be globally ordered such that for all i and t, all variables in $\delta_i(t)$ precede t.

Given these restrictions, each 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 $\{\Delta_1, \ldots, \Delta_n\}$.

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.* $[\hat{v}_j]^{t}_{\ell_j}$ is a value if $t \notin Dom(\delta_i)$.

3.2 Dynamic Semantics

Figure 12 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, $[[\hat{v}_j]^{u}_{\ell_j}]^{\tau}_{\ell_k}$, and the inner embedding, $[\hat{v}_j]^{u}_{\ell_j}$, is a k-value (that is, $u \notin \mathrm{Dom}(\delta_k)$), then the two embeddings can be collapsed into one, $[\hat{v}_j]^{\tau}_{\ell_j:\ell_k}$. 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 argu-

[1] We can encode multiple interfaces to an abstract datatype while satisfying consistency by introducing extra type variables (see the companion technical report).

$$
(\mathbf{1})\quad \frac{e_i \overset{i}{\longmapsto} e_i''}{e_i\ e_i' \overset{i}{\longmapsto} e_i''\ e_i'} \qquad
(\mathbf{2})\quad \frac{e_i' \overset{i}{\longmapsto} e_i''}{v_i\ e_i' \overset{i}{\longmapsto} v_i\ e_i''} \qquad
(\mathbf{3})\quad \frac{e_j \overset{j}{\longmapsto} e_j'}{[e_j]^{\tau}_{\ell_j} \overset{i}{\longmapsto} [e_j']^{\tau}_{\ell_j}}
$$

$$
\begin{array}{lrcl}
(\mathbf{4}) & \mathtt{fix}\ f_i(x_i{:}\tau).e_i & \overset{i}{\longmapsto} & \lambda x_i{:}\tau.\,\{\mathtt{fix}\ f_i(x_i{:}\tau).e_i/f_i\}e_i \\
(\mathbf{5}) & \lambda x_i{:}\tau.\,e_i\ \ v_i & \overset{i}{\longmapsto} & \{v_i/x_i\}e_i \\
& & & \\
(\mathbf{6}) & [c]^{\mathsf{b}}_{\ell_j} & \overset{i}{\longmapsto} & c \\
(\mathbf{7}) & [\hat{v}_j]^{\tau}_{\ell_j} & \overset{i}{\longmapsto} & [\hat{v}_j]^{\bar{\Delta}_i(\tau)}_{\ell_j} \quad (\tau \neq \bar{\Delta}_i(\tau)) \\
(\mathbf{8}) & [[\hat{v}_j]^{u}_{\ell_j}]^{\tau}_{\ell_k} & \overset{i}{\longmapsto} & [\hat{v}_j]^{\tau}_{\ell_j:\ell_k} \quad (u \notin \mathrm{Dom}(\delta_k), \tau = \bar{\Delta}_i(\tau)) \\
& & & \\
(\mathbf{9}) & [\lambda x_j{:}\tau.\,e_j]^{\tau'\to\tau''}_{\ell_j} & \overset{i}{\longmapsto} & \lambda x_i{:}\tau'.\,[\{[x_i]^{\tau}_{i:\mathtt{rev}(\ell_j)}/x_j\}e_j]^{\tau''}_{\ell_j} \quad (x_i \text{ fresh},\ \tau'\to\tau'' = \bar{\Delta}_i(\tau'\to\tau''))
\end{array}
$$

Figure 12: Multiagent Dynamic Semantics

$$
\begin{array}{ll}
& (\lambda y{:}\mathit{int}\to\mathit{int}.\,y\ 3)[\lambda x{:}\mathsf{fh}\to\mathsf{fh}.\,x]^{\mathsf{fh}\to\mathsf{fh}}_{a} \\
(7) & \overset{h}{\longmapsto} (\lambda y{:}\mathit{int}\to\mathit{int}.\,y\ 3)[\lambda x{:}\mathsf{fh}\to\mathsf{fh}.\,x]^{\mathit{int}\to\mathit{int}}_{a} \\
(9) & \overset{h}{\longmapsto} (\lambda y{:}\mathit{int}\to\mathit{int}.\,y\ 3)(\lambda x'{:}\mathit{int}.\,[[x']^{\mathsf{fh}}_{h:a}]^{\mathit{int}}_{a}) \\
(5) & \overset{h}{\longmapsto} (\lambda x'{:}\mathit{int}.\,[[x']^{\mathsf{fh}}_{h:a}]^{\mathit{int}}_{a})3 \\
(5) & \overset{h}{\longmapsto} [[3]^{\mathsf{fh}}_{h:a}]^{\mathit{int}}_{a} \\
(8) & \overset{h}{\longmapsto} [3]^{\mathit{int}}_{h:a:a} \\
(6) & \overset{h}{\longmapsto} 3
\end{array}
$$

Figure 13: Multiagent Evaluation Example

ment 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 Example

As an example, we encode our two-agent calculus by letting δ_h map fh to int and letting δ_a be undefined everywhere. An evaluation which uses all of the novel rules appears in Figure 13. The numbers in the figure are the reduction rules used to take the step. Notice that under this simple system, rules **(7)** and **(9)** encode what was previously "hard-wired" into rule **(H3)**. Similarly, rules **(8)** and **(6)** do the work of **(H4)**.

3.4 Static Semantics

Figure 14 shows the multiagent static semantics for 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 15. 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 is a value with respect to the enclosing agent (for example $[[3]^{\mathsf{t}}_{i}]^{\mathsf{s}}_{j}$ 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 $\delta_i(\mathsf{t}) = \mathit{int}$, $\delta_j(\mathsf{s}) = \mathsf{t}$ and $\delta_k = \emptyset$. Then collapsing the properly typed k-term $[[3]^{\mathsf{t}}_{i}]^{\mathsf{s}}_{j}$ to either $[3]^{\mathsf{s}}_{i}$ or $[3]^{\mathsf{s}}_{j}$ violates the type abstraction properties since neither i nor j knows that s abstracts an *int*. Alternately, if

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

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

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

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

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

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

Figure 14: Multiagent Static Semantics

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

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

Figure 15: Type relations: $\{\Delta\} \vdash \tau \lesssim_{\ell_i} \tau'$

$$\begin{array}{rcl}
\mathit{erase}(x_i) & = & x \\
\mathit{erase}(c) & = & c \\
\mathit{erase}(\lambda x_i{:}\tau.\, e_i) & = & \lambda x{:}\bar{\Phi}(\tau).\, \mathit{erase}(e_i) \\
\mathit{erase}(\texttt{fix}\ f_i(x_i{:}\tau).e_i) & = & \texttt{fix}\ f(x{:}\bar{\Phi}(\tau)).\mathit{erase}(e_i) \\
\mathit{erase}(e_i\ e_i') & = & \mathit{erase}(e_i)\ \mathit{erase}(e_i') \\
\mathit{erase}(\lfloor e_j \rceil^{\tau}_{\ell_j}) & = & \mathit{erase}(e_j)
\end{array}$$

where $\Phi = \bigcup_i \Delta_i$

Figure 16: Multiagent *erase* translation

we were to use sets of agents, instead of (ordered) lists, the reasonable typing rules become too permissive.

3.5 Safety Properties

This section illustrates some of the standard safety theorems of typed programming languages and then discusses abstraction properties that generalize those presented earlier. Rigorous proofs appear in the companion technical report [25].

We begin with type soundness:

Lemma 3.1 (Preservation)
If $\{\Delta\}; \emptyset \vdash_i e_i : \tau$ *and* $e_i \overset{i}{\longmapsto} e_i'$ *then* $\{\Delta\}; \emptyset \vdash_i e_i' : \tau$.

The interesting rules in proving preservation are **(7)** through **(9)** since they rely crucially on the $\lesssim$ relations. In the Appendix, we state the relevant lemmas and present these cases of the proof.

Lemma 3.2 (Progress) *If* e_i *is well-typed then either* e_i *is a value or there exists an* e_i' *such that* $e_i \overset{i}{\longmapsto} e_i'$.

The important point is that rules **(6)** through **(9)** guarantee that $\lfloor v_j \rceil^{\tau}_{\ell_j}$ is not stuck unless it is a value.

Theorem 3.3 (Type Safety) *If* $\{\Delta\}; \emptyset \vdash_i e_i : \tau$ *then there is no stuck* e_i' *such that* $e_i \overset{i}{\longmapsto}{}^{*} e_i'$.

The cost of including embeddings is the addition of several dynamic rules. Worse yet, with recursion and multiple agents, the lists annotating embeddings might grow arbitrarily large. The erasure property stated below essentially shows that these syntactic tricks are only a proof technique.

For erasure to a typed language, it is necessary to use the combined type information of all of the agents. The multiagent definition of *erase* is given in Figure 16, where Φ is the map obtained by taking the union of the compatible Δ_i maps. Informally, $\bar{\Phi}(\tau)$ is the most concrete type for τ that can be found using all n agents' knowledge.

The target language is the simply-typed λ-calculus augmented with `fix`. We have the following lemma:

Lemma 3.4 (Erasure) *If* e_i *is well-typed then either* $e_i \overset{i}{\longmapsto}{}^{*} v_i$ *and* $\mathit{erase}(e_i) \longmapsto^{*} \mathit{erase}(v_i)$, *or* e_i *and* $\mathit{erase}(e_i)$ *both diverge.*

Proof (sketch): By induction on the source derivation. Strengthen the inductive hypothesis to show that for every step of the source language, the erased version takes either zero steps (rules **(6)** through **(9)**) or one step (rules **(4)** and **(5)**). For divergence, show the contrapositive: any term erasing to a non-diverging term is non-diverging. □

We use arguments in the style of subject-reduction to prove safety properties that generalize those of the two-agent case.

Definition 3.5 (Agents(e_i)) *Let* $\underline{\text{Agents}(e_i)}$ *be* i *and the set of agent subscripts appearing in* e_i.

Definition 3.6 (*j*-**free**) *Let e_i be* <u>*j-free*</u> *if $j \notin Agents(e_i)$.*

Definition 3.7 (Oblivious to t) *A set of agents, S, is* <u>*oblivious*</u> *to type t if for all $i \in S$, $\mathsf{t} \notin \mathrm{Dom}(\Delta_i)$ and for all $\mathsf{t'} \neq \mathsf{t}$, $\Delta_i(\mathsf{t'}) \neq \mathsf{t}$.*

Theorem 3.8 (Independence of Evaluation) *Let $\hat{v}_j$ and $\hat{v}'_j$ have type $\bar{\Delta}_j(\mathsf{t})$. If $\{\Delta\}; \emptyset \vdash_i \lambda x_i{:}\mathsf{t}.\, e_i : \mathsf{t} \to b$ and* Agents(e_i) *are oblivious to t then: $(\lambda x_i{:}\mathsf{t}.\, e_i)\ \mathsf{[}\hat{v}_j\mathsf{]}^{\mathsf{t}}_j \longmapsto^* c$ iff $(\lambda x_i{:}\mathsf{t}.\, e_i)\ \mathsf{[}\hat{v}'_j\mathsf{]}^{\mathsf{t}}_j \longmapsto^* c$.*

The proof strengthens the claim to a step-by-step evaluation correspondence when using $\hat{v}_j$ and $\hat{v}'_j$:

Lemma 3.9 (Value Abstraction) *Let $\hat{v}_j$ and $\hat{v}'_j$ have type $\bar{\Delta}_j(\mathsf{t})$. Let $\varphi(e_i)$ mean* Agents(e_i) *are oblivious to t, e_i is j-free, and $\{\Delta\}; [x_i : \mathsf{t}] \vdash_i e_i : \tau$ for some τ. Then if $\varphi(e_i)$ and $\{\mathsf{[}\hat{v}_j\mathsf{]}^{\mathsf{t}}_j / e_i\}$ is not an i-value, then there exists a term e'_i such that:*

- $\{\mathsf{[}\hat{v}_j\mathsf{]}^{\mathsf{t}}_j / x_i\} e_i \xmapsto{i} \{\mathsf{[}\hat{v}_j\mathsf{]}^{\mathsf{t}}_j / x_i\} e'_i$
- $\{\mathsf{[}\hat{v}'_j\mathsf{]}^{\mathsf{t}}_j / x_i\} e_i \xmapsto{i} \{\mathsf{[}\hat{v}'_j\mathsf{]}^{\mathsf{t}}_j / x_i\} e'_i$
- $\varphi(e'_i)$

Proof (sketch): We show that each rule of the operational semantics preserves property φ. □

The generalization of Theorem 2.9 is the following theorem, in which f_h plays the rôle of `ho`. It effectively says that a client containing a value of abstract type t must have obtained that value via a host-provided function.

Theorem 3.10 (Host-Provided Values) *Let h be an agent such that $\bar{\Delta}_h(\mathsf{t}) = \tau_h$. Suppose $\{\Delta\}; \emptyset \vdash_h f_h : \mathsf{b} \to \tau_h$. Let e_i be a term such that* Agents(e_i) *are oblivious to t and $\{\Delta\}; \emptyset \vdash_i \lambda p_i{:}\mathsf{b} \to \mathsf{t}.\, e_i : \tau$. Further suppose $(\lambda p_i{:}\mathsf{b} \to \mathsf{t}.\, e_i)\ \mathsf{[}f_h\mathsf{]}^{\mathsf{b} \to \mathsf{t}}_h \xmapsto{i}^* e'_i$. If v is any value of type t which is a subterm of e'_i, then $v = \mathsf{[}\hat{v}_h\mathsf{]}^{\mathsf{t}}_{\ell_h}$ where $f_h\ \hat{v}'_h \xmapsto{h}^* \hat{v}_h$.*

3.6 Language Extensions

The multiagent language can easily be extended to include new constructs. Products, records, and sums are straightforward to add. To prove the soundness of the rules which propagate embeddings, we need a type relations lemma of the form $\{\Delta\} \vdash \tau^0 \otimes \tau^1 \lesssim_{\ell_i} \tau^2 \otimes \tau^3$ if and only if $\{\Delta\} \vdash \tau^0 \lesssim_{\ell_i} \tau^2$ and $\{\Delta\} \vdash \tau^1 \lesssim_{\ell_i} \tau^3$. Recursive types need a similar lemma that relates a μ-type with its unrolling.

Adding state to the multiagent calculus follows analogously to its addition in the two-agent case. In fact, the symmetry of the multiagent calculus significantly reduces the proliferation of dynamic rules. As noted in Section 2, type soundness does not hold if a reference can be exported at different levels of abstraction. In the multiagent setting, we can succinctly enforce this by extending the definition of Δ_i such that $\Delta_i(\tau \text{ ref}) = \tau \text{ ref}$.

4 Future Work

4.1 Polymorphism and Existential Types

Type abstraction and polymorphism are closely related. Indeed, the example encoding of an agent in Figure 2 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 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.

The typing rule for existential types is similar to that of our embeddings, which is not surprising given that both embeddings and existentials attempt to capture different "views" of a value. Indeed there is a strong connection between abstract types and existentials [12] which may indicate that they are a more natural extension to our language than polymorphic types.

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.

Rule **(9)** essentially keeps track dynamically of which principals are executing in which stack frame. By adding primitives to the language to examine the lists of agents on embeddings, we gain a form of stack inspection, which has been studied in the context of Java security [22, 23]. We intend to investigate the kinds of security properties that can be expressed in such a language.

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 2.9. 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 [20] and Reynolds [19]. Such notions have been incorporated into programming languages such as SML and Haskell [10, 15] 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 Sangiori [16] prove parametricity results for a polymorphic pi-calculus in an operational setting. Rather than add principals to the term language, they use external substitutions to reason about bisimilarity of polymorphic processes in which there are both abstract and concrete views of data values.

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 [14]. 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 Riecke's work on the SLam calculus [8] or Volpano and Smith's work on type-based security [21]. A more syntactic approach to security is taken by Myers [13].

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.

7 Acknowledgments

We thank David Walker, Stephanie Weirich, and the anonymous referees for their helpful comments on earlier drafts of this paper.

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*, pages 267–283, Copper Mountain, CO, December 1995.

[4] Karl Crary. A simple proof technique for certain parametricity results. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*, Paris, France, September 1999. This volume.

[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] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[13] 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.

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

[15] 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.

[16] Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. Technical Report MS-CIS-99-10, University of Pennsylvania, April 1999.

[17] John Hamilton Reppy. *Higher-order Concurrency.* PhD thesis, Cornell University, Ithaca, NY, June 1992. TR 92-1852.

[18] 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.

[19] 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.

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

[21] 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.

[22] Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *Proceedings of 1998 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1998.

[23] Dan Seth Wallach. *A New Approach to Mobile Code Security*. PhD thesis, Princeton University, 1999.

[24] Andrew K. Wright and Mattias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. Preliminary version in Rice TR 91-160.

[25] Steve Zdancewic and Dan Grossman. Syntax and semantics for multiple agents and abstract types. Technical Report 99-1752, Cornell University, March 1999.

A Appendix

We present the most interesting parts of the proof of the Preservation Lemma for the multiagent calculus. We will use the following lemmas; the proofs are in the companion technical report [25].

Lemma A.1 (Idempotency) *For all agents i, and (possibly empty) lists ℓ and ℓ', $\{\Delta\} \vdash \tau \lesssim_{\ell:i:i:\ell'} \tau'$ iff $\{\Delta\} \vdash \tau \lesssim_{\ell:i:\ell'} \tau'$*

Lemma A.2 (Reversal) *If $\{\Delta\} \vdash \tau^0 \lesssim_{\ell_i} \tau^1$ then $\{\Delta\} \vdash \tau^1 \lesssim_{\mathtt{rev}(\ell_i)} \tau^0$*

Lemma A.3 (Arrow Type) *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 $\{\Delta\} \vdash \tau^1 \lesssim_{\ell_i} \tau^3$.*

Lemma A.4 (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 $\{\Delta\}; \Gamma \vdash_i \{e_j/x_j\}e_i : \tau$.*

We recall the statement of Preservation: If $\{\Delta\}; \emptyset \vdash_i e_i : \tau$ and $e_i \overset{i}{\longmapsto} e_i'$ then $\{\Delta\}; \emptyset \vdash_i e_i' : \tau$. The proof is by induction on the derivation that $e_i \overset{i}{\longmapsto} e_i'$ and is by cases on the last rule in the derivation. For formatting reasons, we omit the $\{\Delta\}$ antecedents throughout. Similarly, an omitted environment should be interpreted as $\emptyset$. Here are the cases for the rules peculiar to our calculus:

(6) Trivial.

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

$$\frac{\vdash_j \hat{v}_j : \tau^0 \quad \vdash \tau^0 \lesssim_{\ell_j:i} u}{\vdash_i \lfloor \hat{v}_j \rceil^{u}_{\ell_j} : \bar{\Delta}_i(u)}$$

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

$$\frac{\vdash_j \hat{v}_j : \tau^0 \quad \vdash \tau^0 \lesssim_{\ell_j:i} \bar{\Delta}_i(u)}{\vdash_i \lfloor \hat{v}_j \rceil^{\bar{\Delta}_i(u)}_{\ell_j} : \bar{\Delta}_i(u)}$$

The conclusion is the desired result.

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

$$\frac{\dfrac{\vdash_j \hat{v}_j : \tau^0 \quad \vdash \tau^0 \lesssim_{\ell_j:k} u}{\vdash_k \lfloor \hat{v}_j \rceil^{u}_{\ell_j} : \bar{\Delta}_k(u)} \quad \vdash \bar{\Delta}_k(u) \lesssim_{\ell_k:i} \tau}{\vdash_i \lfloor \lfloor \hat{v}_j \rceil^{u}_{\ell_j} \rceil^{\tau}_{\ell_k} : \bar{\Delta}_i(\tau)}$$

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

$$\frac{\vdash_j \hat{v}_j : \tau^0 \quad \vdash \tau^0 \lesssim_{\ell_j:\ell_k:i} \tau}{\vdash_i \lfloor \hat{v}_j \rceil^{\tau}_{\ell_j:\ell_k} : \bar{\Delta}_i(\tau)}$$

The conclusion is the desired result.

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

$$\frac{[x_j : \tau^0] \vdash_j e_j : \tau^3 \quad \Delta_j(\tau^0) = \tau^0}{\vdash_j \lambda x_j{:}\tau^0.\, e_j : \tau^0 \to \tau^3}$$

$$\frac{\vdash_j \lambda x_j{:}\tau^0.\, e_j : \tau^0 \to \tau^3 \quad \vdash \tau^0 \to \tau^3 \lesssim_{\ell_j:i} \tau^1 \to \tau^2}{\vdash_i \lfloor \lambda x_j{:}\tau^0.\, e_j \rceil^{\tau^1 \to \tau^2}_{\ell_j} : \bar{\Delta}_i(\tau^1 \to \tau^2)}$$

Furthermore, $\Delta_i(\tau^1 \to \tau^2) = \tau^1 \to \tau^2$, so 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 Arrow Type Lemma and the right hand premise of the bottom step, we conclude that $\vdash \tau^0 \lesssim_{\ell_j:i} \tau^1$ and $\vdash \tau^3 \lesssim_{\ell_j:i} \tau^2$. Now notice that the reverse of $\ell_j : i$ is $i : \mathtt{rev}(\ell_j)$ which we shall write $\ell' : j$. So by the Reversal Lemma, $\vdash \tau^1 \lesssim_{\ell':j} \tau^0$. Thus we can derive:

$$\frac{[x_i : \tau^1] \vdash_i x_i : \tau^1 \quad \vdash \tau^1 \lesssim_{\ell':j} \tau^0}{[x_i : \tau^1] \vdash_j \lfloor x_i \rceil^{\tau^0}_{\ell'} : \bar{\Delta}_j(\tau^0)}$$

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

$$\frac{[x_i : \tau^1] \vdash_j \{\lfloor x_i \rceil^{\tau^0}_{\ell'}/x_j\}e_j : \tau_3 \quad \vdash \tau^3 \lesssim_{\ell_j:i} \tau^2}{[x_i : \tau^1] \vdash_i \lfloor \{\lfloor x_i \rceil^{\tau^0}_{\ell'}/x_j\}e_j \rceil^{\tau^2}_{\ell_j} : \bar{\Delta}_i(\tau^2)}$$

$$\frac{[x_i : \tau^1] \vdash_i \lfloor \{\lfloor x_i \rceil^{\tau^0}_{\ell'}/x_j\}e_j \rceil^{\tau^2}_{\ell_j} : \bar{\Delta}_i(\tau^2) \quad \Delta_i(\tau^1) = \tau^1}{\vdash_i \lambda x_i{:}\tau^1.\lfloor \{\lfloor x_i \rceil^{\tau^0}_{\ell'}/x_j\}e_j \rceil^{\tau^2}_{\ell_j} : \tau^1 \to \bar{\Delta}_i(\tau^2)}$$

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