Syntactic Type Abstraction

DAN GROSSMAN, GREG MORRISETT, and STEVE ZDANCEWIC
Cornell University

Software developers often structure programs in such a way that different pieces of code constitute distinct *principals*. Types help define the protocol by which these principals interact. In particular, *abstract types* allow a principal to make strong assumptions about how well-typed clients use the facilities that it provides. We show how the notions of principals and type abstraction can be formalized within a language. Different principals can know the implementation of different abstract types. We use additional syntax to track the flow of values with abstract types during the evaluation of a program and demonstrate how this framework supports syntactic proofs (in the style of subject reduction) for type-abstraction properties. Such properties have traditionally required semantic arguments; using syntax avoids the need to build a model for the language. We present various typed lambda calculi with principals, including versions that have mutable state and recursive types.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures—*Information Hiding*; *Languages*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Syntax*; *Semantics*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Abstract data types*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational Semantics*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type Structure*

General Terms: Languages, Security, Theory, Verification

Additional Key Words and Phrases: Operational semantics, parametricity, proof techniques, syntactic proofs, type abstraction

1. INTRODUCTION

Programmers often use a notion of *principal* when designing the structure of a program. Examples of principals include modules of a large system, a host and its clients, and separate functions. Dividing code into such agents is useful for composing programs. Moreover, with the increasing use of extensible systems, such as Web browsers, databases, and operating systems, this notion of principal becomes critical for reasoning about untrusted clients that interact with host-provided code.

In this paper, we incorporate the idea of principal into various typed lambda calculi. Doing so allows us to formulate security policies and check that the type system enforces them. An example of a useful policy is, "clients can gain access to

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.
Authors' address: Department of Computer Science, 4130 Upson Hall, Cornell University, Ithaca, NY, 14853; email: {danieljg, jgm, zdance}@cs.cornell.edu.

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

Fig. 1. Abstract interface for file handles.

a file handle only through the open procedure."

Consider a host-provided interface for an abstract type of file handles, fh, that includes 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 $\mathsf{fh} = int$, whereas clients do not.

The conventional wisdom is that the use of abstract data types in a type-safe language prevents clients from directly accessing host data. Instead, a client may manipulate such data only via a host-provided interface. To formalize this wisdom, it is necessary to prove theorems that say, "client 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 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 client 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 [Girard et al. 1989], the polymorphic lambda calculus [Reynolds 1974]. The type fh is held abstract by encoding the client as a polymorphic function:

$$\Lambda\mathsf{fh}.\lambda\mathtt{host} : \{\mathtt{open} : string \to \mathsf{fh}, \mathtt{read} : \mathsf{fh} \to char\}.client\_code$$

We can then appeal to Reynolds' parametricity results [Reynolds 1983] to conclude that the client respects the host's interface.

Unfortunately, these representation-independence results are proven using semantic arguments based on a model of the language (see the work of Mitchell [1991], 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 agents with different type information. By "coloring" host code and client code differently, we can track 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 the client code when linked against a host implementation. Figure 2 shows the standard encoding of linking as application. In one step of the standard operational semantics, the host type is substituted throughout the client 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 *client_code*, and all distinctions between principals are lost.

$$\tau = \{\texttt{open} : string \to \mathsf{fh}, \texttt{read} : \mathsf{fh} \to char\}$$

$$(\Lambda \mathsf{fh}.\lambda \texttt{host} : \tau.client\_code) \quad int \quad host\_code$$
$$\longmapsto (\lambda \texttt{host} : \{int/\mathsf{fh}\}\tau.\{int/\mathsf{fh}\}client\_code) \quad host\_code$$
$$\longmapsto \{host\_code/\texttt{host}\}\{int/\mathsf{fh}\}client\_code$$

Fig. 2. Standard encoding for "linking" client and host code.

In the next section, we describe a two-agent setting sufficient for proving interesting properties about the file-handle example. It introduces the general approach of distinguishing principals[1] during evaluation. Section 3 introduces a multiagent calculus that provides for multiple agents, multiple abstract types, and an arbitrary assignment of what types are known to what agents. We prove several safety and abstraction theorems for this calculus; the safety properties for the two-agent calculus follow as corollaries. The next two sections sketch extensions to our system that we believe are more difficult in a less syntactic framework. Section 4 adds references and state; the presentation emphasizes the subtle ways that naive treatments of state can break type abstraction. Section 5 adds recursive types and polymorphism; here we emphasize that these features can be treated as orthogonal to principals or that these features can be encoded using principals. Section 6 surveys related work; this work includes approaches to proving type abstraction, approaches to using syntactic proof techniques, and approaches to putting a notion of principal in a programming language.

This work extends our previously published paper [Zdancewic et al. 1999] by providing a more complete presentation of the two-agent and multiagent calculi and extensions that incorporate state, recursive types, and polymorphism.

2. THE TWO-AGENT CALCULUS

2.1 Syntax

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

Figure 3 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 client terms, C, client values, $\hat{C}$, host terms, H, and host values, $\hat{H}$. The metavariable x_c ranges over client variables, which are disjoint from host variables, ranged over by x_h.

The metavariables b_c and b_h range over values of base type. We assume that the host and client use the same underlying representation for values of base type. These values are thus the fundamental medium of information exchange between agents. For example, the client value 3_c corresponds to the host value 3_h. It would be possible to relax this assumption; conversion between host and client values would then require computation (for instance to change the byte-order of integer

[1]Throughout this paper, we use the words "principal," "agent," and "color" interchangeably.

$$
\begin{array}{rcl}
\tau & ::= & \mathsf{t} \mid \mathsf{b} \mid \tau \to \tau' \\[1em]
C & ::= & x_c \mid b_c \mid \lambda x_c\!:\!\tau.\, C \mid C\ C' \mid \llcorner H \lrcorner_h^\tau \\
\hat{C} & ::= & b_c \mid \lambda x_c\!:\!\tau.\, C \mid \llcorner \hat{H} \lrcorner_h^{\mathsf{t}} \\[1em]
H & ::= & x_h \mid b_h \mid \lambda x_h\!:\!\tau.\, H \mid H\ H' \mid \llcorner C \lrcorner_c^\tau \\
\hat{H} & ::= & b_h \mid \lambda x_h\!:\!\tau.\, H \\[1em]
e & ::= & C \mid H \\
\hat{e} & ::= & \hat{C} \mid \hat{H}
\end{array}
$$

Fig. 3. Two-agent syntax.

values). One of the advantages of distinguishing principals is that places where such marshalling and unmarshalling is needed are made explicit in the program.

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

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 a client term. If the annotation is t, then the client has no information about the form of the term inside the embedding. Thus, an embedding with annotation t and containing a host value is a client value.

A good intuition for the semantics is to imagine two copies of the simply typed lambda calculus augmented with a new type t. In the client copy, t is abstract, whereas in the host copy, t is τ_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. It also means the notion of value depends on the principal and the type information: an embedding with annotation t is a client value. An embedding is never a host value because the annotation is never a type that is abstract to the host.

2.2 Notation

Before describing the semantics, we define some convenient notions. Let e range over both client and host terms, and let $\hat{e}$ range over both client and host values. The *color* of e is c if e is a C term; otherwise, e's color is h. Note that both terms in a syntactically well-formed application are the same color. Because the host and client terms share some semantic rules, we use *polychromatic* rules to range over both kinds of terms. The intention is that all terms mentioned in a polychromatic rule have the same color; 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 alpha conversion, where substituted variables are of the same color. We also define substitution on types, written $\{\tau'/\mathsf{t}\}\tau$. Intuitively, we use the sub-

$$
\begin{array}{llrcll}
\text{Polychromatic Steps} & [P1] & e\ e' & \longmapsto & e''\ e' & \text{if } e \longmapsto e'' \\
 & [P2] & \hat{e}\ e' & \longmapsto & \hat{e}\ e'' & \text{if } e' \longmapsto e'' \\
 & [P3] & (\lambda x\!:\!\tau.\ e)\ \hat{e} & \longmapsto & \{\hat{e}/x\}e & \\
\text{Client Steps} & [C1] & \llcorner H \lrcorner_h^{\tau} & \longmapsto & \llcorner H' \lrcorner_h^{\tau} & \text{if } H \longmapsto H' \\
 & [C2] & \llcorner b_h \lrcorner_h^{\mathsf{b}} & \longmapsto & b_c & \\
 & [C3] & \llcorner \lambda x_h\!:\!\tau.\ H \lrcorner_h^{\tau' \to \tau''} & \longmapsto & \lambda x_c\!:\!\tau'.\ \llcorner \{\llcorner x_c \lrcorner_c^{\tau}/x_h\} H \lrcorner_h^{\tau''} & \\
\text{Host Steps} & [H1] & \llcorner C \lrcorner_c^{\tau} & \longmapsto & \llcorner C' \lrcorner_c^{\tau} & \text{if } C \longmapsto C' \\
 & [H2] & \llcorner b_c \lrcorner_c^{\mathsf{b}} & \longmapsto & b_h & \\
 & [H3] & \llcorner \lambda x_c\!:\!\tau.\ C \lrcorner_c^{\tau' \to \tau''} & \longmapsto & \lambda x_h\!:\!\{\tau_h/\mathsf{t}\}\tau'.\ \llcorner \{\llcorner x_h \lrcorner_h^{\tau}/x_c\} C \lrcorner_c^{\tau''} & \\
 & [H4] & \llcorner \llcorner \hat{H} \lrcorner_h^{\mathsf{t}} \lrcorner_c^{\tau_h} & \longmapsto & \hat{H} &
\end{array}
$$

Fig. 4. Two-agent dynamic semantics.

stitution $\{\tau_h/\mathsf{t}\}\tau$ to produce the host's view of τ.

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

2.3 Dynamic Semantics

Figure 4 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 lambda calculus. The other rules handle embeddings.

Rules [*C1*] and [*H1*] allow evaluation to proceed within embeddings. Inside embeddings, the rules for the other color apply. These "context switches" ensure that terms evaluate with the appropriate rules 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 (see rules [*C2*] and [*H2*]). It is at this point that conversion between data representations would take place.

Rules [*C3*] and [*H3*] maintain the distinction between client and host code. For example, suppose the client contains a host function that is being exported at type $\tau' \to \tau''$. In this case, the client *does* know that the embedding contains a function, so the client can apply it to an argument of a suitable type. If instead the function had been exported at type t, the client would not have been able to apply it. The subtlety is that the host type of the function may be more specific than the client type, such as when $\tau' = \mathsf{t}$.

Thus, [*C3*] converts an embedded host function to a client function with an argument of type τ'. The body of the client version of the function is an embedding of the host code, except that, as the argument now comes from the client, every occurrence of the original argument variable, x_h, is replaced by an embedding of the client's argument variable, $\llcorner x_c \lrcorner_c^{\tau}$. 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" a client value that is an embedded host value. This rule allows the host to recover a value that has been

$$
\begin{array}{ll}
 & (\lambda \mathtt{open}_c : string \to \mathsf{fh}.\ \mathtt{open}_c\ \text{“myfile”}_c)\ \ulcorner \lambda \mathtt{s}_h : string.\ \mathtt{ho}\ \mathtt{s}_h \urcorner_h^{string \to \mathsf{fh}} \\
(1) & (\lambda \mathtt{open}_c : string \to \mathsf{fh}.\ \mathtt{open}_c\ \text{“myfile”}_c)\ (\lambda \mathtt{s}_c : string.\ \ulcorner \mathtt{ho}\ \ulcorner \mathtt{s}_c \urcorner_c^{string} \urcorner_h^{\mathsf{fh}}) \\
(2) & (\lambda \mathtt{s}_c : string.\ \ulcorner \mathtt{ho}\ \ulcorner \mathtt{s}_c \urcorner_c^{string} \urcorner_h^{\mathsf{fh}})\ \text{“myfile”}_c \\
(3) & \ulcorner \mathtt{ho}\ \ulcorner \text{“myfile”}_c \urcorner_c^{string} \urcorner_h^{\mathsf{fh}} \\
(4) & \ulcorner \mathtt{ho}\ \text{“myfile”}_h \urcorner_h^{\mathsf{fh}} \\
 & \vdots \\
(n) & \ulcorner 3_h \urcorner_h^{\mathsf{fh}}
\end{array}
$$

Fig. 5. Client calling `open`.

embedded abstractly in the client. The restricted form of this rule ensures that the term inside the embedding is a value and is itself an embedding. (Specifically, if the annotation is not $\mathtt{t}$, then the term is not a value.) Therefore, no other rule applies. This restriction keeps our system deterministic, a property that we find convenient, but that is probably not necessary.

The crucial point is that any attempt by the client to treat a value of type $\mathtt{t}$ as a function leads to a stuck configuration (no rule applies). 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 lambda calculus, allows us to prove the safety properties of Section 2.6.

2.4 Examples

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

Figure 5 shows the client obtaining a file handle through a host interface. For simplicity, only the host's `open` function is provided to the client. The host implementation, `ho`, takes in a string and produces an integer representing a file handle. This code is embedded inside the client at the more abstract type $string \to \mathsf{fh}$.

Step (1) uses [*C3*] to convert the embedded host function to a client function. Note that the new variable, $\mathtt{s}_c$, is embedded in the host as a client term. Step (2) is a standard β-reduction. Together, these two reduction steps correspond to the linking of host and client code, but, unlike the example in Figure 2, there is no type information about fh given to the client code. Step (3) is another β-reduction, passing in the client 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 [*C1*] allows the host function to proceed. We assume that `ho` returns 3 when applied to “myfile.” This result, embedded within the client code at type fh, is a client value.

The second example (Figure 6) illustrates the client calling the host's `read` function, passing in the file handle $\ulcorner 3_h \urcorner_h^{\mathsf{fh}}$. We assume the host code for `read` is already embedded in the client and exported at the type $\mathsf{fh} \to char$. The body of the host function is `hr`, a host term taking an integer representing a file handle and returning a character read from that file.

In step (1), the client extracts the host function via rule [*C3*]. The type of the argument $\mathtt{handle}_c$ is abstract in the client, so the type annotation is changed to

$$
\begin{array}{ll}
 & \lfloor \lambda \texttt{handle}_h : int.\ \texttt{hr handle}_h \rceil_h^{\mathsf{fh} \to char} \quad \lfloor 3_h \rceil_h^{\mathsf{fh}} \\
(1) & (\lambda \texttt{handle}_c : \mathsf{fh}.\ \lfloor \texttt{hr}\ \lfloor \texttt{handle}_c \rceil_c^{int} \rceil_h^{char}) \quad \lfloor 3_h \rceil_h^{\mathsf{fh}} \\
(2) & \lfloor \texttt{hr}\ \lfloor \lfloor 3_h \rceil_h^{\mathsf{fh}} \rceil_c^{int} \rceil_h^{char} \\
(3) & \lfloor \texttt{hr}\ 3_h \rceil_h^{char} \\
 & \vdots \\
(n) & \lfloor \texttt{'A'}_h \rceil_h^{char} \\
(n+1) & \texttt{'A'}_c
\end{array}
$$

Fig. 6. Client calling `read`.

Polychromatic Rules

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

Client Rules

$$
[Cfn]\ \frac{\Gamma[x_c : \tau'] \vdash C : \tau}{\Gamma \vdash \lambda x_c : \tau'.\ C : \tau' \to \tau} \qquad [HinC]\ \frac{\Gamma \vdash H : \{\tau_h/\mathtt{t}\}\tau}{\Gamma \vdash \lfloor H \rceil_h^{\tau} : \tau}
$$

Host Rules

$$
[Hfn]\ \frac{\Gamma[x_h : \tau'] \vdash H : \tau}{\Gamma \vdash \lambda x_h : \tau'.\ H : \tau' \to \tau}\ \mathtt{t} \notin \tau' \qquad [CinH]\ \frac{\Gamma \vdash C : \tau}{\Gamma \vdash \lfloor C \rceil_c^{\tau} : \{\tau_h/\mathtt{t}\}\tau}
$$

Fig. 7. Two-agent static semantics.

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 client, as a host value. The application ($\texttt{hr}\ 3_h$) proceeds as usual until the host computes the character read from the file. At last, because this embedded character is exported to the client at type *char*, rule [*C2*] produces the value $\texttt{'A'}_c$.

2.5 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 client functions. For host functions, the only difference is that $\mathtt{t}$ is not allowed to appear in the annotation for the argument to a function. Because the host knows that $\mathtt{t} = \tau_h$, this restriction does not limit expressiveness. The convenient effect of this side condition and rule [*CinH*] is that types of host terms never contain $\mathtt{t}$. The presence of $\mathtt{t}$ in host-function types would complicate other rules, such as [*var*] and [*app*], because we would often need to refine a type τ to $\{\tau_h/\mathtt{t}\}\tau$ in order for types to be preserved under evaluation.

The interesting typing rules are those for embeddings. Rule [*HinC*] says that an embedded host term, H, exported to the client 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 $\{\tau_h/\mathtt{t}\}\tau$. In other words, the host may hide type information from the client by replacing some occurrences of τ_h with $\mathtt{t}$ in the exported type. The rule for clients embedded inside of host terms, [*CinH*], is dual in that the host refines the types

$$\begin{array}{rcl} erase(x_i) & = & x \\ erase(b) & = & b \\ erase(\lambda x\!:\!\tau.\, e) & = & \lambda x\!:\!\{\tau_h/\mathsf{t}\}\tau.\, erase(e) \\ erase(e\ e') & = & erase(e)\ erase(e') \\ erase(\llbracket e \rrbracket_i^\tau) & = & erase(e) \end{array}$$

Fig. 8. Two-agent *erase* translation.

provided by the client.

2.6 Safety Properties

In this section, we explore properties of the two-agent calculus including soundness and some type-abstraction theorems. We defer the proofs because they are corollaries to the corresponding theorems in Sections 3.7 and 3.8. These properties are not intended to be as general 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* $\tau = \mathsf{b}$, *then* $\hat{e} = b$ *for some* b.
— *if* $\tau = \tau' \to \tau''$, *then* $\hat{e} = \lambda x\!:\!\tau'.\, e'$ *for some* x *and* e'.
— *if* $\tau = \mathsf{t}$, *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. A term e is <u>*stuck*</u> 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 colors, erase the embeddings, and replace t with τ_h, then we have a well-typed term of the simply typed lambda calculus. Formally, Figure 8 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* $e \longmapsto^* \hat{e}$ *if and only if* $erase(e) \longmapsto^* erase(\hat{e})$.

The interesting fact is that the erasure of rule [*C3*] is basically $\lambda x : \tau.\ e \longmapsto \lambda x' \!:\! \tau.\ \{x'/x\}e$; the substitution of τ_h for t in the erasure process ensures that the types of the bound variables on either side of the transition are the same. Because $\lambda x \!:\! \tau.\ e$ is alpha equivalent to $\lambda x' \!:\! \tau.\ \{x'/x\}e$, the transition [*C3*] corresponds to the identity transition (zero steps) in the erased language.

With soundness and erasure established, we reexamine the abstraction properties of the introduction. Because the host is always capable of providing information to the client, we are particularly interested in evaluations where we assume the host does not do so. Recall that we say *host-free* to describe a client term that has no host terms embedded in it.

One desirable property of the file handle interface is that the client never breaks the type abstraction for file handles. For example, if C is host-free, $\emptyset \vdash \mathit{handle} : \mathsf{fh}$, and $\emptyset \vdash \lambda \mathtt{f_c}\!:\!\mathsf{fh}.\, C : \tau$, then $(\lambda \mathtt{f_c}\!:\!\mathsf{fh}.\, C)\ \mathit{handle}$ never evaluates to a term in which *handle* is treated as an integer. This property is a corollary of type soundness because within the client code the type fh is not equal to *int* (and hence, for example, the expression $\mathtt{f_c} + 3$ is not well-typed).

A stronger property is that the client is oblivious to the particular choice of integer that the host uses to represent a given file handle. More formally:

Theorem 2.7 (Independence of Evaluation). *If* $\ulcorner \hat{H} \urcorner_h^{\mathsf{fh}}$ *and* $\ulcorner \hat{H}' \urcorner_h^{\mathsf{fh}}$ *are well-typed,* C *is host-free, and* $\emptyset \vdash \lambda \mathtt{f}_c\!:\!\mathsf{fh}.\, C : \mathsf{fh} \rightarrow \mathsf{b}$*, then* $(\lambda \mathtt{f}_c\!:\!\mathsf{fh}.\, C)\ \ulcorner \hat{H} \urcorner_h^{\mathsf{fh}} \longmapsto^* b_c$ *if and only if* $(\lambda \mathtt{f}_c\!:\!\mathsf{fh}.\, C)\ \ulcorner \hat{H}' \urcorner_h^{\mathsf{fh}} \longmapsto^* b_c$.

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* C *is host-free,* $[x_c : \mathsf{fh}] \vdash C : \tau$*, and* $\{\ulcorner \hat{H} \urcorner_h^{\mathsf{fh}}/x_c\}C$ *is not a value, then there exists a host-free term* C' *such that:*

— $[x_c : \mathsf{fh}] \vdash C' : \tau$

— $\{\ulcorner \hat{H} \urcorner_h^{\mathsf{fh}}/x_c\}C \longmapsto \{\ulcorner \hat{H} \urcorner_h^{\mathsf{fh}}/x_c\}C'$

— $\{\ulcorner \hat{H}' \urcorner_h^{\mathsf{fh}}/x_c\}C \longmapsto \{\ulcorner \hat{H}' \urcorner_h^{\mathsf{fh}}/x_c\}C'$

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

Theorem 2.9 (File Handles Come from `open`). *Suppose* C *is host-free,* `ho` *is client-free, and* $\lambda \mathtt{open}_c\!:\!\mathit{string} \rightarrow \mathsf{fh}.\, C$ *is well-typed. If* $(\lambda \mathtt{open}_c\!:\!\mathit{string} \rightarrow \mathsf{fh}.\, C)$ *applied to* $\ulcorner \lambda \mathtt{s}_h\!:\!\mathit{string}.\ \mathtt{ho}\ \mathtt{s}_h \urcorner_h^{\mathit{string} \rightarrow \mathsf{fh}}$ *steps to some term* C' *containing* $\ulcorner \hat{H} \urcorner_h^{\mathsf{fh}}$ *as a subterm, then* $\hat{H}$ *was derived from a sequence of the form* $(\mathtt{ho}\ \hat{H}') \longmapsto^* \hat{H}$.

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

3. THE MULTIAGENT CALCULUS

So far, we have described a simple two-agent setting in which the host has strictly more information than the client. We can model many interesting cases in this fashion, but there are times when both principals wish to hide information or in which there are more than two agents involved. For example, we need a multiagent setting to prove safety properties about nested abstract data types.

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

$$
\begin{array}{rrcl}
\text{(agents)} & i,j & \in & \{1,\ldots,n\} \\
\text{(lists)} & \ell & ::= & i \mid i\ell \\
\text{(types)} & \tau & ::= & t \mid \mathsf{b} \mid \tau \rightarrow \tau' \\
(i\text{-terms}) & e_i & ::= & x_i \mid b_i \mid \lambda x_i\!:\!\tau.\ e_i \mid e_i\ e_i' \mid \mathbf{fix}\ f_i(x_i\!:\!\tau).e_i \mid \llcorner e_j \lrcorner_\ell^\tau \\
(i\text{-primvals}) & \hat{v}_i & ::= & b_i \mid \lambda x_i\!:\!\tau.\ e_i \\
(i\text{-values}) & v_i & ::= & \hat{v}_i \mid \llcorner \hat{v}_j \lrcorner_\ell^t\ (t \notin Dom(\delta_i))
\end{array}
$$

Fig. 9. Multiagent syntax.

3.1 Syntax

Figure 9 shows the syntax for the multiagent language. The types include a base type, b, function types, and type variables ranged over by t. In what follows, we also use u and s to range over type variables.

Rather than just two colors of terms, we now assume that there are n agents, where n is fixed. We use the metavariables i, j, and k to range over the set of agents $\{1, \ldots, n\}$.

Every nonembedding term has exactly one color, as indicated by the subscript. Embeddings also have a color that is determined by context—the grammar allows e_i and e_j to produce syntactically identical embedding terms. We explicitly state the color of the term when it is ambiguous or particularly relevant to the discussion. Functions and applications are always constructed from subterms of the same color.

The terms for agent i include variables, x_i, constants, b_i, nonrecursive functions, $\lambda x_i\!:\!\tau.\ e_i$, recursive functions, $\mathbf{fix}\ f_i(x_i\!:\!\tau).e_i$, function applications, $e_i\ e_i'$, and embeddings, $\llcorner e_j \lrcorner_\ell^\tau$. We include both recursive and nonrecursive functions to simplify the dynamic semantics (see rule [*4*] in Figure 10).

An embedding containing a j-term is labeled with a list of agents, ℓ, for reasons explained in Section 3.3. We write simply j for the singleton list containing the agent j, and we use juxtaposition to denote appending two lists ($\ell\ell'$ is the concatenation of lists ℓ and ℓ'). The static semantics requires that the list for an embedded j-term begins with j, so in fact $\ell = j\ell'$ for the term $\llcorner e_j \lrcorner_\ell^\tau$; but we write just ℓ when the j is unimportant. We use $\mathtt{rev}(\ell)$ to mean the list-reversal of ℓ.

3.2 Type Information

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 should not be the case that j thinks $\mathsf{fh} = string$.)

To capture this information, we assume each agent i has a finite partial map from type variables to types called δ_i. To maintain consistency of knowledge among agents, we require that all agents that know the implementation of an abstract type t know the same implementation. We further restrict the δ_i maps so that for each type variable, t, there is a unique, most concrete interpretation of t. For example, we do not allow $\delta_i(t) = t \to t$, or the more subtle $\delta_i(t) = s \to s$, $\delta_j(s) = t$, because these examples do not admit a well-defined notion of most-concrete type for t.

The need for consistency among agents motivates the first part of the following definition; the need for most concrete interpretations of a type motivates the second part.

Definition 3.1. A set $\{\delta_1, \ldots, \delta_n\}$ of finite partial maps from type variables to types is *compatible* if:

—For every $i, j \in \{1, \ldots, n\}$ if $t \in Dom(\delta_i) \cap Dom(\delta_j)$, then $\delta_i(t) = \delta_j(t)$.

—The collection of type variables can be totally ordered such that for every agent, i, and type variable, t, all variables in $\delta_i(t)$ precede t.

Each δ_i extends naturally to a total function, Δ_i from types to types:

$$\begin{aligned}\Delta_i(\mathsf{b}) &= \mathsf{b}\\ \Delta_i(t) &= \begin{cases} t & t \notin Dom(\delta_i) \\ \delta_i(t) & t \in Dom(\delta_i)\end{cases}\\ \Delta_i(\tau \to \tau') &= \Delta_i(\tau) \to \Delta_i(\tau')\end{aligned}$$

Applying Δ_i to a type τ either yields τ (if i has no information about the type variables appearing in τ) or a more concrete version of τ. For example, $\Delta_h(\mathsf{fh} \to \mathsf{fh}) = int \to int$. We say that $\Delta_i(\tau)$ *refines* τ, and we write $\sqsubseteq_i$ for the reflexive, transitive closure of Δ_i viewed as a relation on types. Thus, $\mathsf{fh} \to \mathsf{fh} \sqsubseteq_h int \to int$.

The second condition in the definition of compatibility, and the fact that each agent has a finite amount of type information, guarantees that the process of an agent refining a type halts. That is, the sequence $\tau \sqsubseteq_i \Delta_i(\tau) \sqsubseteq_i \Delta_i(\Delta_i(\tau)) \sqsubseteq_i \ldots$ reaches a fixpoint, which we write $\bar{\Delta}_i(\tau)$. This fixpoint is the most concrete view of τ that agent i is able to determine from its knowledge.

Even if all of the agents pool their information about type variables, the notion of a most concrete view of τ is still valid. We write Φ for the union of the compatible Δ_i maps. It contains the composite knowledge of every agent in the system. Let $\sqsubseteq$ be the reflexive, transitive closure of the relation Φ. As above, the process of refining τ using Φ also terminates: $\tau \sqsubseteq \Phi(\tau) \sqsubseteq \Phi(\Phi(\tau)) \sqsubseteq \ldots$ reaches a fixpoint. The notation $\bar{\Phi}(\tau)$ indicates this fixpoint, which is the most concrete type compatible with τ.

In practice, the compatibility constraints guarantee two things. First, no set of agents can conspire to show that incompatible types are equal (and thus effectively cast an integer to a function, for example). Second, there is a well-defined notion of most concrete type for each agent. The first criterion is clearly necessary for soundness of the system. The second criterion greatly simplifies the static semantics. There may be other notions of compatibility that relax one or both of these conditions while still admitting a sound type system and a well behaved semantics.

3.3 Embeddings

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. That is, $\ulcorner \hat{v}_j \urcorner^t_\ell$ is an i-value if $t \notin Dom(\delta_i)$.

Because an embedding is not an i-primval, a nested embedding (for example, $⦗⦗\lambda x_j : \tau.\ x_j⦘^t_j⦘^s_i$) is never an i-value. We could have made such terms values, but the result would significantly complicate the dynamic semantics: if our example nested embedding were passed to an agent that was able to refine s to an arrow type, then we would need to cross two embedding boundaries to find the function that the agent expects. Instead, our dynamic semantics collapses the two embeddings into one, in this case $⦗\lambda x_j : \tau.\ x_j⦘^s_{ji}$. As a result, values never cross more than one embedding at a time, but the embeddings are now annotated with a list of agents.

Why is this somewhat complicated mechanism necessary? There must be some way of relating the type of a term inside an embedding to the type annotation on the embedding; otherwise, an agent could export an integer as a function. The list of agents allows us to maintain exactly the information needed to establish the connection between the type of the term inside an embedding and the type annotating the embedding. If we forget any agents, inconsistencies may arise. For example, consider three agents, i, j, and k, such that $\delta_i(t) = int$, $\delta_j(s) = t$, and $\delta_k = \emptyset$. Then collapsing the k-term $⦗⦗3_i⦘^t_i⦘^s_j$ to either $⦗3_i⦘^s_i$ or $⦗3_j⦘^s_j$ violates the type-abstraction properties because neither i nor j knows that s abstracts an int. If we use sets of agents instead of (ordered) lists, the reasonable rules become too permissive because we lose the information that agent i must have exported the integer at type t *before* j could export it at type s. As we explain in Section 3.6, the type system admits the nested-embedding term and the collapsed version, $⦗3_i⦘^s_{ij}$, but not the others.

In summary, the lists of agents on embeddings let us remember a particular order of principals, which is necessary for type checking, without treating nested embeddings as values.

3.4 Dynamic Semantics

Figure 10 shows the operational semantics for agent i in the multiagent language. That is, several of the rules depend on the color of the term being reduced, and the rules use i to denote this color.

Rules [*1*], [*2*], [*4*], and [*5*] establish a typical call-by-value semantics.[2] Rule [*3*] allows evaluation inside embeddings. Rule [*6*] lets agent i pull a constant out of an embedding, provided that the constant is exported at type b. This rule corresponds to rules [*C2*] and [*H2*] of the two-agent scenario.

As in the two-agent case, in which the host had more type information than the client, 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, Δ_i captures the type-refinement information available to agent i. Correspondingly, rule [*7*] allows i to refine the type of an embedded value. The side condition on this rule (in conjunction with the conditions on [*8*] and [*9*]) ensures that evaluation is deterministic. This determinism of the type-refinement rules is not critical to the system, but it makes many of the proofs easier because there is only one applicable rule for each evaluation step. The choice of when the operational semantics refines type information interacts with where we must apply the Δ_i maps in the static semantics; we chose to maintain the invariants that the static semantics always derives

[2] We believe the language can easily be adapted to a call-by-name setting.

$$[1]\ \frac{e_i \longmapsto e_i''}{e_i\ e_i' \longmapsto e_i''\ e_i'} \qquad [2]\ \frac{e_i' \longmapsto e_i''}{v_i\ e_i' \longmapsto v_i\ e_i''} \qquad [3]\ \frac{e_j \longmapsto e_j'}{\llcorner e_j \lrcorner_\ell^\tau \longmapsto \llcorner e_j' \lrcorner_\ell^\tau}$$

$$[4] \quad \mathbf{fix}\ f_i(x_i{:}\tau).e_i \ \longmapsto\ \lambda x_i{:}\tau.\ \{\mathbf{fix}\ f_i(x_i{:}\tau).e_i/f_i\}e_i$$

$$[5] \quad (\lambda x_i{:}\tau.\ e_i)\ \ v_i \ \longmapsto\ \{v_i/x_i\}e_i$$

$$[6] \quad \llcorner b_j \lrcorner_\ell^{\mathsf{b}} \ \longmapsto\ b_i$$

$$[7] \quad \llcorner \hat{v}_j \lrcorner_\ell^\tau \ \longmapsto\ \llcorner \hat{v}_j \lrcorner_\ell^{\bar{\Delta}_i(\tau)} \quad (\tau \neq \bar{\Delta}_i(\tau))$$

$$[8] \quad \llcorner\llcorner \hat{v}_j \lrcorner_\ell^u \lrcorner_{k\ell'}^\tau \ \longmapsto\ \llcorner \hat{v}_j \lrcorner_{\ell k \ell'}^\tau \quad (u \notin Dom(\delta_k), \tau = \bar{\Delta}_i(\tau))$$

$$[9] \quad \llcorner \lambda x_j{:}\tau.\ e_j \lrcorner_{j\ell}^{\tau' \to \tau''} \ \longmapsto\ \lambda x_i{:}\tau'.\ \llcorner \{\llcorner x_i \lrcorner_{\mathtt{irev}(\ell)}^\tau / x_j\} e_j \lrcorner_{j\ell}^{\tau''} \ (x_i \text{ fresh},\ \tau' \to \tau'' = \bar{\Delta}_i(\tau' \to \tau''))$$

Fig. 10. Multiagent dynamic semantics: $e \longmapsto e'$ where $color(e) = i$.

the most concrete type for any term, and that types explicitly mentioned in the lambda-abstraction syntax are most concrete. It would be possible to reformulate the calculus so that these conditions are relaxed (by allowing a nondeterministic type-refinement rule in the static semantics), but doing so would require additional proof-normalization arguments.

Rule [*8*] is the multiagent generalization of rule [*H4*]. Both rules allow progress when we have nested embeddings, and the inner embedding cannot be reduced because the inner agent considers it a value. In the two-agent case, this situation can occur only when the outer agent is the host. Because we are not concerned with tracking what values the host manipulates, rule [*H4*] strips away both embeddings. However, in the more symmetric multiagent setting, naively stripping away embeddings loses information about which agents could have contributed information about the type of a term.

Rule [*8*] says that if there are nested embeddings, $\llcorner\llcorner \hat{v}_j \lrcorner_\ell^u \lrcorner_{k\ell'}^\tau$, and the inner embedding, $\llcorner \hat{v}_j \lrcorner_\ell^u$, is a k-value (that is, $u \notin Dom(\delta_k)$), then the two embeddings can be collapsed into one, $\llcorner \hat{v}_j \lrcorner_{\ell k\ell'}^\tau$. Because we append the two lists, we lose no information about which agents have participated in the evaluation of the term.

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 that 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. Because the "inside type" and "outside type" have reversed roles, the list must be in reverse order. Intuitively, the agents that successively provided the function-argument type to i must undo their work in the body of the function. The side condition ensuring that the embedding's type is most concrete with respect to the outer agent preserves the determinism of the semantics and ensures that the

Assume $\delta_h(\mathsf{fh}) = int$ and $\mathsf{fh} \notin Dom(\delta_c)$

$$
\begin{array}{ll}
 & (\lambda y_h : int \to int.\, y_h\ 3_h)\ \llcorner \lambda x_c : \mathsf{fh}.\, x_c \lrcorner_c^{\mathsf{fh}\to\mathsf{fh}} \\
[\mathit{7}] & (\lambda y_h : int \to int.\, y_h\ 3_h)\ \llcorner \lambda x_c : \mathsf{fh}.\, x_c \lrcorner_c^{int\to int} \\
[\mathit{9}] & (\lambda y_h : int \to int.\, y_h\ 3_h)\ (\lambda x'_h : int.\, \llcorner\llcorner x'_h \lrcorner_h^{\mathsf{fh}} \lrcorner_c^{int}) \\
[\mathit{5}] & (\lambda x'_h : int.\, \llcorner\llcorner x'_h \lrcorner_h^{\mathsf{fh}} \lrcorner_c^{int})\ 3_h \\
[\mathit{5}] & \llcorner\llcorner 3_h \lrcorner_h^{\mathsf{fh}} \lrcorner_c^{int} \\
[\mathit{8}] & \llcorner 3_h \lrcorner_{hc}^{int} \\
[\mathit{6}] & 3_h
\end{array}
$$

Fig. 11. Multiagent evaluation example.

explicit type for x_i is most concrete.

3.5 Example

As an example, we encode our two-agent calculus by letting δ_h map fh to int and letting δ_c be undefined everywhere. An evaluation that uses all of the novel rules appears in Figure 11. The host program invokes a client function defined in terms of the abstract type fh. (For simplicity, it is just the identity function.) The evaluation takes place in a host context, enabling the host to apply the function to the value 3 even though the client function expects an argument of type fh. Within the body of the client code, however, the host value 3 is wrapped in an embedding and must be treated abstractly. Even so, the host code is able to recover the value 3 returned by the client. The numbers in the figure are the reduction rules used to take the step. Note 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*].

3.6 Static Semantics

Figure 12 shows the multiagent static semantics. Note that the rules depend on the color of the term and the value of the maps $\delta_1, \ldots, \delta_n$. The rules use i to denote the color of the term. The values of the maps do not change, so we leave them implicit. Therefore, the judgment $\Gamma \vdash e_i : \tau$ should be read as, "Under 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; they are analogous to the side condition on [*Hfn*] in the two-agent case. Similarly, the use of $\bar{\Delta}_i$ in the conclusion of [*embed*] guarantees that an embedded term is viewed by agent i at the most concrete type possible. Formally, the convenient invariant we are maintaining is that $\Gamma \vdash e_i : \tau$ implies $\Delta_i(\tau) = \tau$.

The issue of consistency among agents arises during type checking. For instance, we do not want a principal to export an int as a function. Likewise, we do not want an agent, or collection of agents, to violate the type abstractions represented by the δ_i maps. Thus, we need to relate the type of the expression inside the embedding to the type annotation on the embedding.

$$[const]\ \frac{}{\Gamma \vdash b_i : \mathsf{b}} \qquad [var]\ \frac{}{\Gamma \vdash x_i : \Gamma(x_i)} \qquad [app]\ \frac{\Gamma \vdash e_i : \tau' \to \tau \quad \Gamma \vdash e'_i : \tau'}{\Gamma \vdash e_i\ e'_i : \tau}$$

$$[abs]\ \frac{\Gamma[x_i : \tau'] \vdash e'_i : \tau \quad \Delta_i(\tau') = \tau'}{\Gamma \vdash \lambda x_i {:} \tau'.\, e'_i : \tau' \to \tau}\ (x_i \notin Dom(\Gamma))$$

$$[\mathit{fix}]\ \frac{\Gamma[f_i : \tau' \to \tau][x_i : \tau'] \vdash e'_i : \tau \quad \Delta_i(\tau' \to \tau) = \tau' \to \tau}{\Gamma \vdash \mathbf{fix}\ f_i(x_i{:}\tau').e'_i : \tau' \to \tau}\ (f_i, x_i \notin Dom(\Gamma), f_i \neq x_i)$$

$$[embed]\ \frac{\Gamma \vdash e_j : \tau' \quad \vdash \tau' \lesssim_{j\ell i} \tau}{\Gamma \vdash \llbracket e_j \rrbracket^{\tau}_{j\ell} : \bar{\Delta}_i(\tau)}$$

Fig. 12. Multiagent static semantics: $\Gamma \vdash e_i : \tau$.

$$[eq]\ \frac{\bar{\Delta}_i(\tau) = \bar{\Delta}_i(\tau')}{\vdash \tau \lesssim_i \tau'} \qquad [trans]\ \frac{\vdash \tau \lesssim_\ell \tau'' \quad \vdash \tau'' \lesssim_{\ell'} \tau'}{\vdash \tau \lesssim_{\ell\ell'} \tau'}$$

Fig. 13. Type relations: $\vdash \tau \lesssim_\ell \tau'$.

We establish an agent-list indexed family of relations on types, $\tau \lesssim_\ell \tau'$. Judgments of the form $\vdash \tau \lesssim_\ell \tau'$, showing when two types may be related by the list ℓ, are given in Figure 13.[3] These rules say that $\tau^0 \lesssim_{i_1 i_2 \ldots i_m} \tau^m$ if there exist types $\tau^1, \ldots, \tau^{m-1}$ such that agent i_k is able to show that $\tau^{k-1} = \tau^k$ for $k \in \{1, \ldots, m\}$. Informally, the agents are able to chain together their knowledge of type information to show that $\tau^0 = \tau^m$.

The $[embed]$ rule uses the $\lesssim_{j\ell i}$ 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.

3.7 Safety Properties

This section illustrates some of the standard safety theorems of typed programming languages and then presents an embedding-erasure transformation that commutes with evaluation.

The following standard lemmas help establish type soundness. The proofs are straightforward.

Lemma 3.2 (Canonical Forms). *Assuming $\emptyset \vdash v_i : \tau$,*

— *if $\tau = \mathsf{b}$, then $v_i = b_i$ for some b.*

— *if $\tau = \tau' \to \tau''$, then $v_i = \lambda x_i {:} \tau'.\, e'_i$ for some x_i and e'_i.*

— *if $\tau = t$, then $t \notin Dom(\delta_i)$ and $v_i = \llbracket \hat{v}_j \rrbracket^t_{j\ell}$ for some $\hat{v}_j$ and ℓ.*

[3]We give a nondeterministic rule for $[trans]$ because we are not concerned with an algorithmic presentation of type checking. This formulation lets us slightly simplify the proofs of the Type-Relations Properties, but is not essential to their correctness.

LEMMA 3.3 (SUBSTITUTION). *Suppose* $\Gamma[x_j : \tau'] \vdash e_i : \tau$ *and* $\Gamma \vdash e_j : \tau'$. *Then* $\Gamma \vdash \{e_j/x_j\}e_i : \tau$.

We also need several properties about the $\lesssim$ relations.

LEMMA 3.4 (TYPE-RELATION PROPERTIES).

Idempotence: $\vdash \tau \lesssim_{\ell i i \ell'} \tau'$ *if and only if* $\vdash \tau \lesssim_{\ell i \ell'} \tau'$.

Reversal: *If* $\vdash \tau \lesssim_{\ell} \tau'$, *then* $\vdash \tau' \lesssim_{\texttt{rev}(\ell)} \tau$.

Arrow: *If* $\vdash \tau^1 \to \tau^2 \lesssim_{\ell} \tau^3 \to \tau^4$, *then* $\vdash \tau^1 \lesssim_{\ell} \tau^3$ *and* $\vdash \tau^2 \lesssim_{\ell} \tau^4$.

Proofs of the first two properties are simple arguments by induction on the derivation of the type relation. Our proof of the last property requires a tedious normalization argument showing that if $\vdash \tau^1 \to \tau^2 \lesssim_{\ell} \tau^3 \to \tau^4$, then there is a derivation that relates only types whose top-level constructors are arrows. We give a brief sketch of the argument: the proof is by induction on the length of the list ℓ. The base case, when ℓ is of length one, follows from the compatibility of the δ_i relations. The inductive case is straightforward if the types in the chain are all arrows, so we suppose that in a chain of types showing $\vdash \tau^1 \to \tau^2 \lesssim_{\ell} \tau^3 \to \tau^4$ the first type variable encountered is t. Then the type before t must be some $\tau^5 \to \tau^6$. After some number of type variables in the chain, we must again have some arrow type. From the consistency conditions on the δ_i maps, this next arrow type must be $\tau^5 \to \tau^6$. Hence we did not need the occurrence of t in the chain, and can simply replace it with $\tau^5 \to \tau^6$. In other words, any occurrences of type variables provide no additional information, so we can remove them, reducing the problem to a chain consisting only of arrow types.

We now have the results we need to prove the two main lemmas.

LEMMA 3.5 (PRESERVATION). *If* $\emptyset \vdash e_i : \tau$ *and* $e_i \longmapsto e_i'$, *then* $\emptyset \vdash e_i' : \tau$.

PROOF. By induction on the derivation that $e_i \longmapsto e_i'$. We proceed by cases on the last step of the derivation. Cases [*1*], [*2*], and [*3*] follow from induction. Cases [*4*] and [*5*] follow immediately from the Substitution Lemma. Case [*6*] is trivial to prove. We consider the remaining cases individually:

[*7*] We have a typing derivation as follows:

$$\frac{\emptyset \vdash \hat{v}_j : \tau'' \quad \vdash \tau'' \lesssim_{j\ell i} \tau'}{\emptyset \vdash \llbracket \hat{v}_j \rrbracket_{j\ell}^{\tau'} : \bar{\Delta}_i(\tau')}$$

Because $\bar{\Delta}_i(\tau')$ equals $\bar{\Delta}_i(\bar{\Delta}_i(\tau'))$, by [*eq*] we have that $\vdash \tau' \lesssim_i \bar{\Delta}_i(\tau')$. Thus by [*trans*] we have that $\vdash \tau'' \lesssim_{j\ell i i} \bar{\Delta}_i(\tau')$, and we can remove the second i by Idempotence. We can now derive:

$$\frac{\emptyset \vdash \hat{v}_j : \tau'' \quad \vdash \tau'' \lesssim_{j\ell i} \bar{\Delta}_i(\tau')}{\emptyset \vdash \llbracket \hat{v}_j \rrbracket_{j\ell}^{\bar{\Delta}_i(\tau')} : \bar{\Delta}_i(\tau')}$$

The conclusion is the desired result.

[*8*] We have a typing derivation as follows:

$$\dfrac{\dfrac{\emptyset \vdash \hat{v}_j : \tau'' \quad \vdash \tau'' \lesssim_{j\ell k} u}{\emptyset \vdash \llcorner \hat{v}_j \lrcorner^u_{j\ell} : \bar{\Delta}_k(u)} \quad \vdash \bar{\Delta}_k(u) \lesssim_{k\ell' i} \tau'}{\emptyset \vdash \llcorner\llcorner \hat{v}_j \lrcorner^u_{j\ell} \lrcorner^{\tau'}_{k\ell'} : \bar{\Delta}_i(\tau')}$$

The fact that the inner embedding is labeled with a list starting with j follows from the Canonical Forms Lemma. Furthermore, $u \notin \Delta_k$, so $\bar{\Delta}_k(u) = u$. Thus by [*trans*] and the premises, we have $\vdash \tau'' \lesssim_{j\ell k k \ell' i} \tau'$. So Idempotence proves that $\vdash \tau'' \lesssim_{j\ell k \ell' i} \tau'$. We can now derive:

$$\dfrac{\emptyset \vdash \hat{v}_j : \tau'' \quad \vdash \tau'' \lesssim_{j\ell k\ell' i} \tau'}{\emptyset \vdash \llcorner \hat{v}_j \lrcorner^{\tau'}_{j\ell k\ell'} : \bar{\Delta}_i(\tau')}$$

The conclusion is the desired result.

[*9*] We have a typing derivation as follows:

$$\dfrac{\dfrac{[x_j : \tau^0] \vdash e_j : \tau^3 \quad \Delta_j(\tau^0) = \tau^0}{\emptyset \vdash \lambda x_j\!:\!\tau^0.\, e_j : \tau^0 \to \tau^3} \quad \vdash \tau^0 \to \tau^3 \lesssim_{j\ell i} \tau^1 \to \tau^2}{\emptyset \vdash \llcorner \lambda x_j\!:\!\tau^0.\, e_j \lrcorner^{\tau^1 \to \tau^2}_{j\ell} : \bar{\Delta}_i(\tau^1 \to \tau^2)}$$

Furthermore, the concreteness side condition on rule [*9*] implies that $\Delta_i(\tau^1 \to \tau^2) = \tau^1 \to \tau^2$. So by the definition of Δ_i, we know that $\Delta_i(\tau^1) = \tau^1$ and $\Delta_i(\tau^2) = \tau^2$. From the Arrow Lemma and the right-hand premise of the bottom step, we conclude that $\vdash \tau^0 \lesssim_{j\ell i} \tau^1$ and $\vdash \tau^3 \lesssim_{j\ell i} \tau^2$. The reverse of $j\ell i$ is $i\,\mathtt{irev}(\ell) j$, so by the Reversal Lemma, $\vdash \tau^1 \lesssim_{i\mathtt{rev}(\ell) j} \tau^0$. Thus we can derive:

$$\dfrac{\overline{[x_i : \tau^1] \vdash x_i : \tau^1} \quad \vdash \tau^1 \lesssim_{i\mathtt{rev}(\ell) j} \tau^0}{[x_i : \tau^1] \vdash \llcorner x_i \lrcorner^{\tau^0}_{i\mathtt{rev}(\ell)} : \bar{\Delta}_j(\tau^0)}$$

Because the original derivation provides $\Delta_j(\tau^0) = \tau^0$, we conclude $[x_i : \tau^1] \vdash \llcorner x_i \lrcorner^{\tau^0}_{i\mathtt{rev}(\ell)} : \tau^0$. It also provides $[x_j : \tau^0] \vdash e_j : \tau^3$. Because x_i is fresh, we may weaken this claim to $[x_i : \tau^1][x_j : \tau^0] \vdash e_j : \tau^3$. Hence by Substitution, $[x_i : \tau^1] \vdash \{\llcorner x_i \lrcorner^{\tau^0}_{i\mathtt{rev}(\ell)} / x_j\} e_j : \tau^3$. So we can derive:

$$\dfrac{\dfrac{[x_i : \tau^1] \vdash \{\llcorner x_i \lrcorner^{\tau^0}_{i\mathtt{rev}(\ell)} / x_j\} e_j : \tau^3 \quad \vdash \tau^3 \lesssim_{j\ell i} \tau^2}{[x_i : \tau^1] \vdash \llcorner \{\llcorner x_i \lrcorner^{\tau^0}_{i\mathtt{rev}(\ell)} / x_j\} e_j \lrcorner^{\tau^2}_{j\ell} : \bar{\Delta}_i(\tau^2)} \quad \Delta_i(\tau^1) = \tau^1}{\emptyset \vdash \lambda x_i\!:\!\tau^1.\, \llcorner \{\llcorner x_i \lrcorner^{\tau^0}_{i\mathtt{rev}(\ell)} / x_j\} e_j \lrcorner^{\tau^2}_{j\ell} : \tau^1 \to \bar{\Delta}_i(\tau^2)}$$

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

Lemma 3.6 (Progress). *If e is well-typed, then either e is a value or there exists an e' such that $e \longmapsto e'$.*

Proof Sketch. The proof is a straightforward inductive argument on the structure of e. The interesting case is when $e = e_i = \llcorner v_j \lrcorner^{\tau}_{\ell}$. If $\Delta_i(\tau) \neq \tau$, then rule [*7*] applies. Else, if v_j is an embedding, then rule [*8*] applies. Otherwise, we proceed

$$\begin{array}{rcl}
erase(x_i) & = & x \\
erase(b_i) & = & b \\
erase(\lambda x_i{:}\tau.\, e_i) & = & \lambda x{:}\bar{\Phi}(\tau).\, erase(e_i) \\
erase(\textbf{fix}\ f_i(x_i{:}\tau).e_i) & = & \textbf{fix}\ f(x{:}\bar{\Phi}(\tau)).erase(e_i) \\
erase(e_i\ e_i') & = & erase(e_i)\ erase(e_i') \\
erase(\llcorner e_j \lrcorner_\ell^\tau) & = & erase(e_j)
\end{array}$$

$$\text{where } \Phi = \bigcup_i \Delta_i$$

Fig. 14. Multiagent *erase* translation.

by cases on the form of τ. If $\tau = \mathsf{b}$, rule [*6*] applies. If $\tau = \tau' \to \tau''$, rule [*9*] applies. Else e_i is a value. □

Given the previous two lemmas, we conclude type safety:

THEOREM 3.7 (TYPE SAFETY). *If* $\emptyset \vdash e : \tau$, *then there is no stuck* e' *such that* $e \longmapsto^* e'$.

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 combine the type information of all the agents. The multiagent definition of *erase* is given in Figure 14, where we recall that Φ is the map obtained by taking the union of the compatible Δ_i maps. $\bar{\Phi}(\tau)$ is the most concrete type for τ that can be found using all n agents' knowledge.

The target language has one agent and no embeddings.

LEMMA 3.8 (ERASURE). *If* e_i *is well-typed then either both* e_i *and* $erase(e_i)$ *diverge or* $e_i \longmapsto^* v_i$ *and* $erase(e_i) \longmapsto^* erase(v_i)$.

PROOF SKETCH. By induction on the number of steps in the source derivation. For one step, show that if $e_i \longmapsto e_i'$, then either $erase(e_i) = erase(e_i')$ or $erase(e_i) \longmapsto erase(e_i')$. That is, the erased version takes either zero steps (if the source step uses one of rules [*6*] through [*9*]) or one step (if the source step uses rule [*4*] or [*5*]). For divergence, show the contrapositive—any term erasing to a nondiverging term is nondiverging. The essence of the argument is that the source derivation can take only a finite number of steps before using rules [*4*] or [*5*]. □

3.8 Type-Abstraction Properties

In this section, we use subject-reduction arguments to prove type-abstraction properties that generalize those of the two-agent case.

Definition 3.9. Let $\underline{\text{Agents}(e_i)}$ be the set of colors appearing in e_i. (The set necessarily includes i as well as any agents appearing in lists annotating embedding subterms of e_i.)

Definition 3.10. A set of agents, S, is ***oblivious*** to type t if for all $i \in S$, $t \notin Dom(\delta_i)$.

THEOREM 3.11 (INDEPENDENCE OF EVALUATION). *Let e_i be a term such that* $\text{Agents}(e_i)$ *are oblivious to t and* $\emptyset \vdash \lambda x_i\!:\!t.\, e_i : t \to \mathsf{b}$. *Let $\hat{v}_j$ and $\hat{v}'_j$ be closed, well-typed terms with type $\bar{\Delta}_j(t)$. Then $(\lambda x_i\!:\!t.\, e_i)\ ⌈\hat{v}_j⌉^t_j \longmapsto^* b_i$ if and only if $(\lambda x_i\!:\!t.\, e_i)\ ⌈\hat{v}'_j⌉^t_j \longmapsto^* b_i$.*

First note that $\hat{v}_j$ and $\hat{v}'_j$ are *primitive* values. Hence $⌈\hat{v}_j⌉^t_j$ and $⌈\hat{v}'_j⌉^t_j$ are values. Also, we know $j \notin \text{Agents}(e_i)$: because $\hat{v}_j$ is a constant or a function and $⌈\hat{v}_j⌉^t_j$ is well-typed, [*embed*] ensures that $t \in \mathit{Dom}(\delta_j)$. $\text{Agents}(e_i)$ are oblivious to t, so $j \notin \text{Agents}(e_i)$.

The proof of the theorem strengthens the claim to a step-by-step evaluation correspondence when using $\hat{v}_j$ and $\hat{v}'_j$. We give the intuition before presenting the formal lemma: we maintain that the intermediate terms in the two evaluation sequences are always exactly the same except that every occurrence of $\hat{v}_j$ in one is replaced with $\hat{v}'_j$ in the other. To ensure this correspondence, we show that every occurrence of $\hat{v}_j$ is safely within an embedding with a type that agents other than j can relate to t. These agents are oblivious to t, so the embedding's type must be at least as abstract as t. We also maintain that there are no j-terms other than the ones we are abstracting and that j appears nowhere in an embedding list except as the first element. These conditions suffice to argue that j never helps the other agents break the type abstraction.

More specifically, properties (1) and (2) of φ (defined in the lemma) suffice to show that $\text{Agents}(e_i) \setminus \{j\}$ cannot distinguish values of type t without the help of agent j; properties (3), (4), and (5) suffice to show that j does not provide such help.

LEMMA 3.12 (VALUE ABSTRACTION). *Let $\hat{v}_j$ and $\hat{v}'_j$ have type $\bar{\Delta}_j(t)$. Let $\varphi(e_i)$ mean:*

(*1*) $[x_j : \bar{\Delta}_j(t)] \vdash e_i : \tau$ *for some* τ.
(*2*) $\text{Agents}(e_i) \setminus \{j\}$ *are oblivious to* t.
(*3*) *The only j-term in e_i is x_j. (It may appear zero, one, or more times.)*
(*4*) *If $⌈x_j⌉^\tau_{j\ell}$ is a k-term in e_i, then $\vdash t \lesssim_{\ell k} \tau$.*
(*5*) *j never appears in an embedding list within e_i except as the first element.*

Then if $\varphi(e_i)$ and $\{\hat{v}_j/x_j\}e_i \longmapsto e^1_i$, then there exists an e^2_i such that:

(a) $\varphi(e^2_i)$
(b) $\{\hat{v}_j/x_j\}e^2_i =_\alpha e^1_i$
(c) $\{\hat{v}'_j/x_j\}e_i \longmapsto \{\hat{v}'_j/x_j\}e^2_i$

PROOF SKETCH. By induction on the derivation of $\{\hat{v}_j/x_j\}e_i \longmapsto e^1_i$, proceeding by cases on the last rule used in the derivation.

Rules [*1*], [*2*], and [*3*] essentially follow from induction and the definition of substitution. Note that for [*3*], the embedded term is not a j-term because the only j-term, $\hat{v}_j$, is a value.

Rules [*4*] and [*5*] follow from the definition of substitution and the fact that the variable for which we substitute a term must be distinct from x_j because the term and x_j are different colors. Furthermore, $\hat{v}_j$ is closed, so the substitution has no effect on it.

Rule [*6*] is trivial except that we must prove the term inside the embedding is not $\hat{v}_j$ (otherwise $e_i^1 = \hat{v}_i$, so property (c) will not hold). Assume for contradiction the term inside is $\hat{v}_j$. Then by property (3), before the substitution this term was x_j. Then by properties (4) and (5), a j-free list of agents, ℓ, can show $\vdash t \lesssim_\ell \mathsf{b}$. A simple proof by induction on the rules for $\lesssim_\ell$ shows that this conclusion contradicts property (2).

Rules [*7*] and [*8*] are straightforward to verify. Note that in [*8*] the outer agent list cannot contain j because of properties (3) and (5).

Rule [*9*] requires an analogous argument to the one we used in [*6*] to show that the term inside the embedding cannot be x_j. Hence the agent list on the embedding does not contain j, so property (5) is preserved even though the agent list is reversed. The substitution in the body of the function is unproblematic because $\hat{v}_j$ is closed and therefore unaffected. □

This lemma suffices to prove Independence of Evaluation because, given the assumptions of the theorem, $\varphi((\lambda x_i{:}t.\ e_i)\ \llcorner x_j \lrcorner_j^t)$. By induction on the length of the evaluation, property (b) holds for b_i, where b_i is the result of the evaluation with $\hat{v}_j$. This implies that the same b_i must be the result of the evaluation with $\hat{v}'_j$ because the two results are alpha equivalent.

The generalization of Theorem 2.9 is the following theorem. It effectively says that a client containing a value of abstract type fh must have obtained that value via a host-provided function.

THEOREM 3.13 (HOST-PROVIDED VALUES). *Suppose* $\mathtt{prog}$ *is a closed and well-typed term of the form* $(\lambda \mathtt{open}_i : \mathsf{b} \to \mathsf{fh}.\ \mathtt{client})\llcorner \lambda x_h : \mathsf{b}.\ \mathtt{ho}\lrcorner_h^{\mathsf{b}\to\mathsf{fh}}$. *Further suppose* Agents($\mathtt{client}$) *are oblivious to* fh*,* h *knows* fh*,* $\mathtt{ho}$ *is embedding-free, and* $\mathtt{prog} \longmapsto^* e_i$*. Then any subterm of* e_i *that is a closed value of type* fh *has the form* $\llcorner \hat{v}_h \lrcorner_{h\ell}^{\mathsf{fh}}$*, and there exists an* e_h *such that* $\{e_h/x_h\}\mathtt{ho} \longmapsto^* \hat{v}_h$ *and* $erase(e_h) = b$ *for some base value* b*.*

Note that it is not necessarily the case that $\{b_h/x_h\}\mathtt{ho} \longmapsto^* \hat{v}_h$ because $\bar{\Delta}_h(\mathsf{fh})$ could be a function type and because the dynamic semantics does not evaluate under functions. Also note that we make no restrictions on the agents in $\mathtt{client}$ and their type abstractions except that none of them know the implementation of fh. For example, one agent could export fh more abstractly to another agent.

The proof consists of an intricate subject-reduction argument, so we first present the general idea. Intuitively, the only h terms are $\mathtt{ho}$ and the result of applying $\mathtt{ho}$ to an argument. Because only h knows fh, all values of type fh come from applying $\mathtt{ho}$. The main complication is tracking all the functions that are essentially $\lambda x_h : b.\ \mathtt{ho}$ except that they differ in color and embeddings: The first step of $\mathtt{prog}$ uses [*9*] to convert $\llcorner \lambda x_h : \mathsf{b}.\ \mathtt{ho}\lrcorner_h^{\mathsf{b}\to\mathsf{fh}}$ to $\lambda x_i : \mathsf{b}.\ \llcorner\{\llcorner x_i\lrcorner_i^{\mathsf{b}}/x_h\}\mathtt{ho}\lrcorner_h^{\mathsf{fh}}$, so we must consider the latter term a legitimate producer of file handles. Agent i could pass this term to another agent using [*9*] again, and that result must also be legitimate.

Due to these complications, we carefully state a property that evaluation of a well-typed term preserves: Figure 15 presents two relations, $\varphi(e)$ and $\psi(e)$, where as usual, we mean the least relations closed under the appropriate inference rules. In the rules, we assume the color of e does not know fh; we use colors j and k to

$$[A]\ \frac{}{\varphi(b)} \qquad [B]\ \frac{}{\varphi(x)}$$

$$[C]\ \frac{\varphi(e) \quad \varphi(e')}{\varphi(e\ e')} \qquad [D]\ \frac{\varphi(e)}{\varphi(\lambda x\!:\!\tau.\ e)} \qquad [E]\ \frac{\varphi(e)}{\varphi(\mathbf{fix}\ f(x\!:\!\tau).e)} \qquad [F]\ \frac{\varphi(e) \quad h \notin \ell}{\varphi(\ulcorner e \urcorner_{\ell}^{\tau})}$$

$$[G]\ \frac{\{e_h/x_h\}\mathtt{ho} \longmapsto^* e'_h \quad erase(e_h) = b \quad \emptyset \vdash e_h : \mathsf{b} \quad \vdash \mathsf{fh} \lesssim_{\ell'} \tau \quad h \notin \ell\ell'}{\varphi(\ulcorner e'_h \urcorner_{h\ell}^{\tau})}$$

$$[H]\ \frac{\psi(e)}{\varphi(e)} \qquad [I]\ \frac{}{\psi(\lambda x_j\!:\!\mathsf{b}.\ \ulcorner\{\ulcorner x_j \urcorner_j^{\mathsf{b}}/x_h\}\mathtt{ho}\urcorner_h^{\mathsf{fh}})}$$

$$[J]\ \frac{\psi(\lambda y_k\!:\!\tau^1.\ \ulcorner e \urcorner_{\ell}^{\tau^2}) \quad \vdash \tau^3 \lesssim_{j\mathtt{rev}(\ell')} \tau^1 \quad h \notin \ell'}{\psi(\lambda x_j\!:\!\tau^3.\ \ulcorner\ulcorner\{\ulcorner x_j \urcorner_{j\mathtt{rev}(\ell')}^{\tau^1}/y_k\}e\urcorner_{\ell}^{\tau^2}\urcorner_{k\ell'}^{\tau^4})}$$

$$[K]\ \frac{}{\psi(\ulcorner \lambda x_h\!:\!\mathsf{b}.\ \mathtt{ho}\urcorner_h^{\mathsf{b}\to\mathsf{fh}})}$$

Fig. 15. Host-Provided Values preservation property.

range over any such color. We implicitly assume the context of the Host-Provided Values Theorem, so $\lambda x_h\!:\!\mathsf{b}.\ \mathtt{ho}$ is closed, well-typed, and embedding-free.

Rules $[A]$ through $[F]$ are just structural rules; we use them to "find the important terms," namely the functions that are essentially $\lambda x_h : b.\ \mathtt{ho}$ and the results of applying such functions. No other h-terms are allowed.

Rule $[G]$ accepts terms that are the (intermediate) results of applying such functions. Because such functions differ from $\mathtt{ho}$ in terms of embeddings, $[G]$ does not require that $e = \ulcorner e'_h \urcorner_{hl}^{\tau}$ where there exists a v_h such that $\{v_h/x_h\}\mathtt{ho} \longmapsto^* e'_h$. Rather, v_h may be a nonvalue e_h so long as its erasure is a value.

Rule $[H]$ accepts the terms that are essentially $\lambda x_h\!:\!b.\ \mathtt{ho}$ by using the auxiliary relation ψ. Rule $[I]$ accepts the function we have after one step of $\mathtt{prog}$. For each time that the function is passed to another agent via rule $[9]$, we use the rule $[J]$ to conclude that the resulting function is still acceptable. We include rule $[K]$ so that $\varphi(\mathtt{prog})$ holds. It is easy to show that $\psi(e)$ implies e is closed and well-typed.

It is straightforward to prove that if $\varphi(e)$ is preserved under evaluation then the Host-Provided Values Theorem is true: first verify that $\varphi(\mathtt{prog})$. Therefore, $\varphi(e_i)$ for any e_i such that $\mathtt{prog} \longmapsto^* e_i$. A simple induction on the derivation of $\varphi(e_i)$, appealing to the canonical forms of values as necessary, suffices to prove that for any subterm v that is a closed value of type fh, it must be that $\varphi(v)$. Rule $[G]$ is necessary to derive $\varphi(v)$, and the antecedents of this rule are strong enough to prove the theorem. (A separate inductive argument shows that if $\psi(e)$, then no subterm of e is a closed value of type fh.)

We now proceed with the rigorous proof that φ is preserved. As usual, the subject-reduction argument relies on a substitution lemma that we prove before presenting the main lemma.

Lemma 3.14 (Host-Provided Substitution). *If $\varphi(e)$, $\varphi(e')$ and $color(e') = color(x)$, then $\varphi(\{e'/x\}e)$.*

Proof. The proof is by induction on the derivation of $\varphi(e)$, proceeding by cases on the last rule in the derivation. Cases $[A]$ and $[B]$ are trivial. Cases $[C]$, $[D]$, $[E]$, $[F]$ follow from straightforward inductive arguments. Case $[G]$ follows because $\{e_h/x_h\}\texttt{ho}$ is closed and evaluation preserves this property. Case $[H]$ follows because $\psi(e)$ implies that e is closed (as shown by induction on the derivation of $\psi(e)$). □

Lemma 3.15 (Host-Provided Preservation). *If $\varphi(e)$, $\emptyset \vdash e : \tau$, and $e \longmapsto e'$, then $\varphi(e')$.*

Proof. The proof is by induction on the derivation of $e \longmapsto e'$, proceeding by cases on the last rule in the derivation.

[*1*] Follows from the induction hypothesis and $[C]$.

[*2*] Follows from the induction hypothesis and $[C]$.

[*3*] The derivation of $\varphi(e)$ ends with $[F]$ or $[G]$. (Note that $[H]$ is impossible because $\psi(e)$ holds only for values or embeddings of values, which do not fit the conditions of [*3*].) If $[F]$, then the result follows from the induction hypothesis and $[F]$. If $[G]$, then we know $e = \llcorner e'_h \lrcorner^{\tau}_{h\ell}$ and $\{e_h/x_h\}\texttt{ho} \longmapsto^n e'_h$ for some n and appropriate e_h. Therefore, we know that $\{e_h/x_h\}\texttt{ho} \longmapsto^{n+1} e''_h$ and $e' = \llcorner e''_h \lrcorner^{\tau}_{h\ell}$. So we can use $[G]$ to derive $\varphi(e')$.

[*4*] Follows from the induction hypothesis, $[E]$, Host-Provided Substitution, and $[D]$.

[*5*] Then $e = (\lambda x_j : \tau'.\ e'')\ v_j$, v_j has type τ', and the derivation ends with $[C]$. Therefore, $\varphi(v_j)$ and $\varphi(\lambda x_j : \tau'.\ e'')$; the derivation of the latter ends in $[D]$ or $[H]$. If $[D]$, then the result follows from the induction hypothesis, $[D]$, and Host-Provided Substitution. If $[H]$, then $\psi(\lambda x_j : \tau'.\ e'')$, the derivation ending in $[I]$ or $[J]$. Therefore, e'' has the form $\llcorner e''' \lrcorner^{\tau''}_{\ell}$. It suffices to show that $\varphi(\{v_j/x_j\}\llcorner e''' \lrcorner^{\tau''}_{\ell})$. To prove this fact, we prove that $\psi(\lambda x_j : \tau'.\ \llcorner e''' \lrcorner^{\tau''}_{\ell})$, $\varphi(v_j)$, and v_j has type τ' implies two stronger facts:

—$\mathit{erase}(v_j) = b$ for some b

—For any e_j of type τ', if there is a b such that $\varphi(e_j)$ and $\mathit{erase}(e_j) = b$, then $\varphi(\{e_j/x_j\}\llcorner e''' \lrcorner^{\tau''}_{\ell})$.

It is clear that this strengthened claim suffices. We prove it by induction on the derivation of $\psi(\lambda x : \tau'.\ \llcorner e''' \lrcorner^{\tau''}_{\ell})$, proceeding by cases on the last rule used.

For case $[I]$, $\tau' = \mathsf{b}$, so the first fact follows from the Canonical Forms Lemma and the well-typedness of e. For the second fact, $\{e_j/x_j\}\llcorner e''' \lrcorner^{\tau''}_{\ell} = \llcorner \{\llcorner e_j \lrcorner^{\mathsf{b}}_{j}/x_h\}\texttt{ho} \lrcorner^{\mathsf{fh}}_{h}$. So $[G]$ applies by letting $e_h = \llcorner e_j \lrcorner^{\mathsf{b}}_{j}$ and $e'_h = \{e_h/x_h\}\texttt{ho}$.

For case $[J]$, we have $\lambda x_j : \tau'.\ \llcorner e''' \lrcorner^{\tau''}_{\ell} = \lambda x_j : \tau^3.\ \llcorner\llcorner \{\llcorner x_j \lrcorner^{\tau^1}_{\texttt{jrev}(\ell')}/y_k\} e^1 \lrcorner^{\tau^2}_{\ell} \lrcorner^{\tau^4}_{j\ell'}$. Furthermore, we have from the derivation that $\psi(\lambda y_k : \tau^1.\ \llcorner e^1 \lrcorner^{\tau^2}_{\ell})$ and $\vdash \tau^3 \lesssim_{\texttt{jrev}(\ell')} \tau^1$. From the induction hypothesis, we conclude that if some v_k has type τ^1 and $\varphi(v_k)$, then $\mathit{erase}(v_k) = b$ for some b. Because $\vdash \tau^3 \lesssim_{\texttt{jrev}(\ell')} \tau^1$, we can use the Canonical Forms Lemma to conclude that if v_j has type τ^3, then $\mathit{erase}(v_j) = b$, which is our first obligation. From the induction hypothesis, we know that for any e_k of type τ^1 such that $\varphi(e_k)$ and $\mathit{erase}(e_k) = b$, we know $\varphi(\{e_k/y_k\}\llcorner e^1 \lrcorner^{\tau^2}_{\ell})$. So for arbitrary e_j such that $\varphi(e_j)$ and $\mathit{erase}(e_j) = b$, we let $e_k = \llcorner e_j \lrcorner^{\tau^1}_{\texttt{jrev}(\ell')}$ to conclude that $\varphi(\{\llcorner e_j \lrcorner^{\tau^1}_{\mathtt{jrev}(\ell')}/y_k\}\llcorner e^1 \lrcorner^{\tau^2}_{\ell})$. Therefore, from the definition of substitution, we have $\varphi(\llcorner\{e_j/x_j\}\{\llcorner x_j \lrcorner^{\tau^1}_{\mathtt{jrev}(\ell')}/y_k\}e^1\lrcorner^{\tau^2}_{\ell})$. Finally, because $h \notin \ell'$, we can use rule $[F]$ and the definition of substitution to derive $\varphi(\{e_j/x_j\}\llcorner\llcorner\{\llcorner x_j \lrcorner^{\tau^1}_{\mathtt{jrev}(\ell')}/y_k\}e^1\lrcorner^{\tau^2}_{\ell}\lrcorner^{\tau^4}_{j\ell'}$, which is our second obligation.

[*6*] Immediate because $\varphi(b)$.

[*7*] Then the derivation of $\varphi(e)$ ends with $[F]$ or $[G]$. (Note that $[H]$ is impossible because that derivation would have to end with $[K]$, but no nonhost agent can refine $\mathsf{b} \to \mathsf{fh}$.) For either $[F]$ or $[G]$, inspection of the antecedents reveals that the same rule applies after the step.

[*8*] Then each of the last two steps of the derivation of $\varphi(e)$ are $[F]$ or $[G]$. (Note that $[H]$ is impossible because $[K]$ is not a nested embedding and $\mathsf{b} \to \mathsf{fh}$ is not a type variable.) If both are $[F]$, then the induction hypothesis and one use of $[F]$ let us conclude $\varphi(e')$. If the last step is $[F]$ and the second-to-last step is $[G]$, then the induction hypothesis and one use of $[G]$ let us conclude $\varphi(e')$. Note that the well-typedness of e suffices to satisfy the requirement that the type on the annotation be more abstract than fh. The remaining cases have $[G]$ as the last step; we show by contradiction this situation is impossible. Suppose the derivation of $\varphi(e)$ ends in $[G]$. Then the e'_h in the consequent of the rule is an h-value of the form $\llcorner \hat{v}_j \lrcorner^{u}_{\ell}$ and $u \notin \delta_h$. But $\mathtt{ho}$ is embedding-free and $\emptyset \vdash e_h : \mathsf{b}$, so it is easy to show that $\{e_h/x_h\}\mathtt{ho}$ cannot evaluate to a value that is an embedding.

[*9*] Then the derivation of $\varphi(e)$ ends in $[H]$ or $[F]$. (Note that $[G]$ is impossible because Agents($\mathtt{client}$) are oblivious to fh, so τ must be at least as abstract as fh.) If the derivation ends in $[H]$, we know $e = \llcorner\lambda x_h : \mathsf{b}.\ \mathtt{ho}\lrcorner^{\mathsf{b}\to\mathsf{fh}}_{h}$ and therefore $e' = \lambda x_i : \mathsf{b}.\ \llcorner\{\llcorner x_i\lrcorner^{\mathsf{b}}_{i}/x_h\}\mathtt{ho}\lrcorner^{\mathsf{fh}}_{h}$, so $[I]$ lets us conclude $\varphi(e')$. If the derivation ends with $[F]$, we know the second-to-last step is $[D]$ or $[H]$. If $[D]$, then the result follows from Host-Provided Substitution, $[F]$, and $[D]$. If $[H]$, then we know $\psi(\lambda y : \tau^1.\ \llcorner e''\lrcorner^{\tau^2}_{\ell'})$ where $e = \llcorner\lambda y : \tau^1.\ \llcorner e''\lrcorner^{\tau^2}_{\ell'}\lrcorner^{\tau^3\to\tau^4}_{\ell}$. (Either $[I]$ or $[J]$ could be the last rule used to derive this result; e has this form in both cases.) Therefore, we can use $[J]$ and $[H]$ to derive $\varphi(e')$. Note that the well-typedness of e and the fact that $h \notin \ell$ suffice to satisfy the antecedents of $[J]$. $\square$

It is worth considering how we could generalize this proof and proof technique. Generalizing the means by which clients obtain file handles is straightforward. The function $\mathtt{ho}$ is just an example of a constructor; it is an expression that is exported at a type in which fh appears in a positive position. By tracking these constructors, we can show that all abstract values originate from them. More generally, using syntax and subject reduction to track the flow of values requires that we can always "find" the interesting terms, which is the purpose of rules $[A]$ through $[F]$. Rule $[G]$ describes the property of interest (in this case that the term came from an invocation of the host implementation of open). The auxiliary predicate ψ describes how the host term can appear after being exported to any number of agents—embeddings can pile up, but only in a uniform way which does not change the underlying code, only the interface at which the code is available.

4. REFERENCES AND STATE

In this section, we augment the program states of the preceding development with a mutable store and appropriate expressions for manipulating the store. The goal is to extend our type-abstraction results to a stateful setting without being unduly restrictive.

The next section is a brief overview of our formalization of state, ignoring issues regarding agents and type abstraction. The point is simply to establish basic notation for the discussion that follows. Our work follows closely the standard treatment of references in a subject-reduction setting, e.g., see the work of Wright and Felleisen [1994] or Harper [1994]. We then give an informal description of the issues that make the store a particularly interesting addition to our framework. Then we formally augment our calculus with state and discuss the reasons for particular design decisions. Finally, we augment the proofs of type soundness and safety properties appropriately.

4.1 Notation

Ignoring types and agents, a store, M, is a partial map from labels to values. The collection of labels is an infinite set; we use l to range over its members. A program state is a pair of a store and a closed expression, and the dynamic semantics operates on program states. Three new expression forms allow a program to manipulate the store. Reference creation, **ref** v, evaluates to a fresh label, l, and extends the store to map l to v. Dereference, $!l$ evaluates to $M(l)$ where M is the current store. Update, $l := v$, evaluates to v and changes the current store to map l to v.

So that we may give appropriate types to labels, we add a unary type constructor **ref**. Under store M, the values of type **ref** τ are those l such that $M(l)$ has type τ. The static semantics ensures that well-formed program states refer only to labels defined in the store. This restriction applies to the store itself, not just to the closed expression. Hence, stores and expressions are well-formed relative what labels are in the store and what types these labels have. Store types, ranged over by Ψ, formalize these assumptions as a partial map from labels to types.

Figure 16 summarizes the preceding considerations by presenting the syntax and semantics for a lambda calculus with state. Note that the rule for store well-formedness permits the store to contain cycles.

4.2 Problems with State

Although the semantics associated with our stateful calculus is well understood, discussing type abstraction between principals in such a setting is considerably more complicated. Here we present examples of the difficulties.

The first difficulty is that the expressions that a program may use in its evaluation are not syntactically captured by the subterms of an expression. That is, we previously described some client expression, C, as *host-free*, but now we must account for C obtaining host code from the store. For example, in the expression $(!l\ \llbracket v \rrbracket_h^{\mathbf{t}})$, whether or not the evaluation is independent of the value of v depends on the value of $M(l)$ where M is the current store. An obvious solution is to define "host-free" over the entire program state (M, C), but this solution is unnecessarily restrictive. Instead, our safety theorems use a notion of expressions *reachable* from C in (M, C).

Syntax

$$\begin{array}{rcl}
\tau & ::= & \mathsf{b} \mid \tau \to \tau' \mid \mathbf{ref}\,\tau \\
e & ::= & v \mid x \mid e\ e' \mid \mathbf{fix}\ f(x{:}\tau).e \mid \mathbf{ref}\ e \mid e := e' \mid\ !e \\
v & ::= & b \mid l \mid \lambda x{:}\tau.\ e
\end{array}$$

Dynamic Semantics

$$\frac{(M, e_1) \longmapsto (M', e_1')}{(M, e_1\ e_2) \longmapsto (M', e_1'\ e_2)} \qquad \frac{(M, e_2) \longmapsto (M', e_2')}{(M, v_1\ e_2) \longmapsto (M', v_1\ e_2')}$$

$$\frac{(M, e_1) \longmapsto (M', e_1')}{(M, e_1 := e_2) \longmapsto (M', e_1' := e_2)} \qquad \frac{(M, e_2) \longmapsto (M', e_2')}{(M, v_1 := e_2) \longmapsto (M', v_1 := e_2')}$$

$$\frac{(M, e) \longmapsto (M', e')}{(M, !e) \longmapsto (M', !e')} \qquad \frac{(M, e) \longmapsto (M', e')}{(M, \mathbf{ref}\ e) \longmapsto (M', \mathbf{ref}\ e')}$$

$$\frac{}{(M, (\lambda x{:}\tau.\ e)\ v) \longmapsto (M, \{v/x\}e)} \qquad \frac{}{(M, !l) \longmapsto (M, M(l))}$$

$$\frac{}{(M, \mathbf{ref}\ v) \longmapsto (M[l \mapsto v], l)}\ l \text{ fresh} \qquad \frac{}{(M, l := v) \longmapsto (M[l \mapsto v], v)}$$

$$\frac{}{(M, \mathbf{fix}\ f(x{:}\tau).e) \longmapsto (M, \lambda x{:}\tau.\ \{\mathbf{fix}\ f(x{:}\tau).e/f\}e)}$$

Static Semantics

$$\frac{\vdash M : \Psi \quad \Psi; \emptyset \vdash e : \tau}{\vdash (M, e) : \tau} \qquad \frac{Dom(M) = Dom(\Psi) \quad \forall l \in Dom(M).\ \Psi; \emptyset \vdash M(l) : \Psi(l)}{\vdash M : \Psi}$$

$$\frac{}{\Psi; \Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Psi; \Gamma \vdash b : \mathsf{b}} \qquad \frac{}{\Psi; \Gamma \vdash l : \mathbf{ref}\,\Psi(l)}$$

$$\frac{\Psi; \Gamma \vdash e : \tau' \to \tau \quad \Psi; \Gamma \vdash e' : \tau'}{\Psi; \Gamma \vdash e\ e' : \tau} \qquad \frac{\Psi; \Gamma[x : \tau'] \vdash e : \tau}{\Psi; \Gamma \vdash \lambda x{:}\tau'.\ e : \tau' \to \tau}\ x \notin Dom(\Gamma)$$

$$\frac{\Psi; \Gamma[f : \tau' \to \tau][x : \tau'] \vdash e : \tau}{\Psi; \Gamma \vdash \mathbf{fix}\ f(x{:}\tau').e : \tau' \to \tau}\ f, x \notin Dom(\Gamma), f \neq x \qquad \frac{\Psi; \Gamma \vdash e : \mathbf{ref}\,\tau}{\Psi; \Gamma \vdash !\, e : \tau}$$

$$\frac{\Psi; \Gamma \vdash e : \mathbf{ref}\,\tau \quad \Psi; \Gamma \vdash e' : \tau}{\Psi; \Gamma \vdash e := e' : \tau} \qquad \frac{\Psi; \Gamma \vdash e : \tau}{\Psi; \Gamma \vdash \mathbf{ref}\ e : \mathbf{ref}\,\tau}$$

Fig. 16. Lambda calculus with state.

The second difficulty is that the store provides a new medium for interagent communication. For example, the client could assign to a shared reference that the host then dereferences. Suppose that l is a shared reference and that the program evaluates the client program state $(M, l := \ulcorner v \urcorner^{\mathsf{t}}_h)$. If we interpret this program to mean that $M(l)$ is now $\ulcorner v \urcorner^{\mathsf{t}}_h$, then, from the host's perspective, the store now contains a client term. If we mean $M(l)$ is now $\ulcorner\ulcorner v \urcorner^{\mathsf{t}}_h \urcorner^{\tau_h}_c$, then the store now contains a nonvalue. The dual case is also a potential problem: if the host assigns v of type τ_h to l and then the client dereferences l, the result must be a client expression that preserves the type abstraction.

In general, when a reference is assigned, we do not know which agents will dereference the value. Despite this lack of knowledge, the semantics must create the correct embeddings to allow a syntactic account of type abstraction. Fortunately, because all expressions are colored, we do know which agent last assigned to a reference.

The third difficulty is that the host should not provide the client the same reference as a $\mathbf{ref}\,\mathsf{t}$ and a $\mathbf{ref}\,\tau_h$. Using a **let** notation as syntactic sugar for function application, an expression that violates this requirement could have the form

$$\begin{array}{l}\mathbf{let}\ r_h = \mathbf{ref}\ v\\ \mathbf{let}\ \mathit{client} = [\lambda x_c\!:\!\mathbf{ref}\,\mathsf{t}.\ \lambda y_c\!:\!\mathbf{ref}\,\tau_h.\ e]_c^{\mathbf{ref}\,\tau_h\to\mathbf{ref}\,\tau_h\to\tau}\\ \mathit{client}\ r_h\ r_h\end{array}$$

To see the problem, consider what the values of $!x_c$ and $!y_c$ should be in the body of the client function, e, after the application $\mathit{client}\ r_h\ r_h$. An answer that preserves the types of the expressions is $!x_c = [v]_h^{\mathsf{t}}$ and $!y_c = [v]_h^{\tau_h}$ respectively, but this answer requires that the dynamic semantics depends on the types of x_c and y_c. This dependence is undesirable in the substitution-based semantics we use. This approach is even less sensible if we consider a code fragment within e such as:

$$\begin{array}{l}\mathbf{let}\ \_ = y_c := (v' : \tau_h)\\ !x_c\end{array}$$

Because the agent knows τ_h, it can construct a value v' of this type and assign it to the shared reference. The ensuing dereference cannot preserve the type of $!x_c$ unless it is something like $[v']_h^{\mathsf{t}}$, but v' is not a host-provided term.

Even if we could somehow resolve the subject-reduction issues exemplified above, the resulting system can still break type abstraction. The example we give below is well-known in the context of the ML programming language; the same device works in that setting [Pierce and Sangiorgi 1999]. If the client term e can determine that values of types $\mathbf{ref}\,\mathsf{t}$ and $\mathbf{ref}\,\tau_h$ are the same value, then it can correctly conclude that $\mathsf{t} = \tau_h$. The following *client* fragment makes such a determination using aliasing information. We assume only that the client can distinguish two values of type τ_h, call them v_1 and v_2.

$$\begin{array}{l}\mathbf{let}\ \_ = y_c := v_1\\ \mathbf{let}\ abs_1 = !x_c\\ \mathbf{let}\ \_ = y_c := v_2\\ \mathbf{let}\ \_ = x_c := abs_1\\ \mathbf{if}\ !y_c == v_1\\ \mathbf{then}\,\text{“aha, } \mathsf{t} = \tau_h\text{”}\end{array}$$

Given these problems, the challenge is to prohibit an agent from viewing the contents of the same reference at two different levels of abstraction. In so doing, we should not otherwise restrict the language's expressiveness.

4.3 The Revised Calculus

Before presenting the formal calculus, we summarize how we solve the problems just outlined.

The definition of *host-free* is straightforward: an expression e' is reachable from (M, e) if e' is a subterm of e or l is a subterm of e and e' is reachable from $(M, M(l))$.[4] (M, e) is host-free if all terms reachable from (M, e) are client terms.

To correctly handle interagent communication through the store, we make the semantics of the dereference operator depend on the color of the expression currently stored in the reference. For example, if $M(l) = v_c$, then $(M, !l_h) \longmapsto (M, \llcorner v_c \lrcorner_c^{\tau})$ because l_h and v_c are different colors. (To know what type τ to put on the embedding, we annotate labels with types—see below.) Conversely, $(M, !l_c) \longmapsto (M, v_c)$. This technique requires only that every expression (even an embedding) has one color and that we can determine this color at run time. Intuitively, a value is embedded precisely when it is received by a different agent.

Alternatively, we could have made all dereference operations return embeddings. Allowing i-embeddings in i code is perfectly reasonable—in fact, our multiagent calculus does not prevent it. We choose not to take this approach because we prefer to introduce embeddings precisely when interagent communication occurs. Although our theorems do not rely on the fact that intraagent evaluation does not introduce embeddings, we find this fact aesthetically pleasing. (If crossing an embedding boundary requires computation to change data representation, avoiding these redundant embeddings reduces computational overhead.)

Finally, we prevent an agent from exporting a reference at different levels of abstraction. To enforce this restriction syntactically, we associate a type expression with each label, l_i^{τ}, and reference creation expression, $\mathbf{ref}_{\tau}\ e$. Such expressions have type $\mathbf{ref}\ \tau$ and must be exported at precisely $\mathbf{ref}\ \tau$, or at a fully abstract type, such as t. So the host must decide when creating a reference at what level of abstraction it will expose the contents of the reference. To prevent exposing a reference at a different level of abstraction, the rules for relating the type on an embedding annotation to the type of the embedded value do not substitute for any types under the **ref** constructor.

Of course, the host could allow the client to access a reference of type $\mathbf{ref}\ \mathsf{t}$ by also providing functions $get = \llcorner \lambda x_h : \mathbf{ref}\ \mathsf{t}.\ !x_h \lrcorner_h^{\mathbf{ref}\ \mathsf{t} \to \tau_h}$ and $set = \llcorner \lambda x_h : \tau_h.\ \lambda y_h : \mathbf{ref}\ \mathsf{t}.\ y_h := x_h \lrcorner_h^{\mathsf{t} \to \mathbf{ref}\ \mathsf{t} \to \mathsf{t}}$. It might appear, then, that the client could use *get* and *set* to encode the aliasing example above that breaks type abstraction. However, because we assume that the client does not know the function bodies associated with *get* and *set*, the client cannot ever conclude for sure that two references of different types are aliases. After all, *get* could perform other computation that happened to evaluate to the contents of the supposedly aliased value. In other words, the correctness of the aliasing device relies on the atomic, completely specified behavior of the assignment and dereference operators.

Having given some intuition for our design decisions, we now present the augmented calculus. We extend the multiagent calculus because the symmetry of this calculus actually simplifies much of the presentation, but we explain the system in terms of the familiar host and client example.

First, recall that δ_i is a finite partial map from type variables to types; it encodes what type variables agent i knows. (So in the two-agent case, δ_h maps t to τ_h and

[4] Because we are generally concerned with stores of finite size, we interpret this definition inductively; otherwise, a coinductive interpretation is warranted.

$$[!1]\quad \frac{}{(M, \mathbf{ref}_\tau\ v_i) \longmapsto (M[l^\tau \mapsto v_i], l_i^\tau)}\ l^\tau \text{fresh} \qquad [!2]\quad \frac{}{(M, l_i^\tau := v_i) \longmapsto (M[l^\tau \mapsto v_i], v_i)}$$

$$[!3]\quad \frac{\text{color}(M(l^\tau)) = i}{(M, !l_i^\tau) \longmapsto (M, M(l^\tau))} \qquad [!4]\quad \frac{\text{color}(M(l^\tau)) = j \neq i}{(M, !l_i^\tau) \longmapsto (M, \llcorner M(l^\tau) \lrcorner_j^\tau)}$$

$$[!5]\quad \frac{(M, e_j) \longmapsto (M, e'_j)}{(M, \llcorner e_j \lrcorner_\ell^\tau) \longmapsto (M, \llcorner e'_j \lrcorner_\ell^\tau)} \qquad [!6]\quad \frac{}{(M, \llcorner l_j^\tau \lrcorner_\ell^{\mathbf{ref}\,\tau}) \longmapsto (M, l_i^\tau)}$$

Fig. 17. Dynamic semantics with state (partial): $(M, e) \longmapsto (M', e')$ where $color(e) = i$.

δ_c is the empty map.) We extend δ_i to a function Δ_i on types that does not operate under the **ref** constructor:

$$\begin{aligned}
\Delta_i(\mathsf{b}) &= \mathsf{b} \\
\Delta_i(t) &= \begin{cases} t & t \notin Dom(\delta_i) \\ \delta_i(t) & t \in Dom(\delta_i) \end{cases} \\
\Delta_i(\tau \to \tau') &= \Delta_i(\tau) \to \Delta_i(\tau') \\
\Delta_i(\mathbf{ref}\,\tau) &= \mathbf{ref}\,\tau
\end{aligned}$$

Hence a value of type $\mathbf{ref}\,\tau$ is exported either abstractly or as $\mathbf{ref}\,\tau$, but not at $\mathbf{ref}\,\tau'$ for $\tau' \neq \tau$.

In the two-agent case, we previously wrote Δ_h as $\{\tau_h/\mathsf{t}\}\tau$ and Δ_c as the identity function. Using syntax that suggests substitution is now misleading because substitution does not occur under **ref**. As before, $\bar{\Delta}_i(\tau) = \bigsqcup_{n \geq 0} \Delta_i^n(\tau)$, which in the one-abstract-type case is just Δ_i.

The dynamic semantics is a straightforward combination of the multiagent semantics presented in Figure 10 and the stateful semantics presented in Figure 16. Specifically, the system is defined by

—Rules [*4*]–[*9*] in Figure 10, suitably extended so that they operate on program states and do not modify the store.

—The top six rules in Figure 16, which allow evaluation to proceed in subexpressions of applications and state operations.

—Rules for evaluating the state operations and allowing evaluation within an embedding, as shown in Figure 17.

Rules [!*3*] and [!*4*] enforce our decision to introduce embeddings when the heap facilitates interagent communication. Rule [!*6*] allows an agent to extract a label from an embedding when the agent has sufficient type information to know that the embedding contains a label. Note that the type on the label is τ, and the type on the embedding is $\mathbf{ref}\,\tau$; this equality is consistent with our definition of Δ_i.

The interesting additions to the static semantics appear in Figure 18. To obtain the full semantics, include the rules presented in Figures 12 and 13 with the modification that all typing judgments include the store type Ψ in the context.

Store contents are typechecked according to their color. Hence a reference may hold a value of any color. In the two-agent setting, this policy means that if l^{t} contains a host value, then that value must have type τ_h under the host's static semantics; but if the reference contains an agent value, then the value must have

$$[heap1]\quad \frac{\vdash M : \Psi \quad \Psi;\emptyset \vdash e_i : \tau}{\vdash (M, e) : \tau}$$

$$[heap2]\quad \frac{\begin{array}{l} Dom(M) = Dom(\Psi) \\ \forall l^\tau \in Dom(M).\quad (\Psi;\emptyset \vdash M(l^\tau) : \tau' \quad \bar{\Delta}_i(\tau) = \tau' \quad \text{color}(M(l^\tau)) = i) \end{array}}{\vdash M : \Psi}$$

$$[label]\quad \frac{l^\tau \in Dom(\Psi)}{\Psi;\Gamma \vdash l_i^\tau : \mathbf{ref}\,\tau} \qquad [ref]\quad \frac{\Psi;\Gamma \vdash e_i : \tau' \quad \bar{\Delta}_i(\tau) = \tau'}{\Psi;\Gamma \vdash \mathbf{ref}_\tau\ e_i : \mathbf{ref}\,\tau}$$

$$[assign]\quad \frac{\Psi;\Gamma \vdash e_i : \mathbf{ref}\,\tau \quad \Psi;\Gamma \vdash e_i' : \tau' \quad \bar{\Delta}_i(\tau) = \tau'}{\Psi;\Gamma \vdash e_i := e_i' : \tau'} \qquad [deref]\quad \frac{\Psi;\Gamma \vdash e_i : \mathbf{ref}\,\tau}{\Psi;\Gamma \vdash !e_i : \bar{\Delta}_i(\tau)}$$

Fig. 18. Static semantics with state (partial).

type t under the agent's static semantics. Note this treatment requires that we can determine the color of $M(l^\tau)$.

We maintain the convenient invariant that $\Psi;\Gamma \vdash v_i : \tau$ implies $\tau = \Delta_i(\tau)$. Because references have the type with which they are annotated, this invariant means the type of the dereference operation must explicitly be $\bar{\Delta}_i(\tau)$ where the reference has type $\mathbf{ref}\,\tau$. The rules for reference creation and assignment require the subexpression that evaluates to the reference's eventual contents to have type $\bar{\Delta}_i(\tau)$. In other words, i must know that τ is a legitimate abstraction of τ', which by our invariant is most concrete.[5] For example, in our two-agent system, consider $\mathbf{ref}_{\mathsf{t}}\ v$ where v has type τ_h. This expression is well-formed if it is a host term, but not if it is an agent term.

The rule for relating the type on an embedding to the type of the embedded expression prevents a $\mathbf{ref}\,\tau$ from being exported at any reference type other than $\mathbf{ref}\,\tau$ even if $\mathbf{ref}\,\tau$ is a constituent of another type. This important fact follows from the definition of the $\lesssim_\ell$ relation, which is defined in terms of the Δ_i maps. In the two-agent case, the right antecedent of rule [*embed*] is simply $\Delta_h(\tau) = \tau'$ for host terms and $\tau = \tau'$ for agent terms.

4.4 Safety Properties

The type soundness and erasure results that we established for the calculi without state extend naturally to the stateful calculus. Therefore, we only present the aspects of the theorems and proofs that pertain to the additional rules.

The Canonical Forms Lemma now describes the form of v_i based on τ, assuming that $\Psi;\emptyset \vdash v_i : \tau$. The addition is that if $\tau = \mathbf{ref}\,\tau'$, then v_i has the form $l_i^{\tau'}$.

For the sake of type soundness, the Substitution Lemma still need describe only substitution over expressions, as opposed to program states. This simplicity is because the dynamic semantics substitutes through only expressions. As we will see, however, the Independence of Evaluation Lemma needs definitions for substitution through program states and well-formedness of open stores (that is, stores that may contain values with free variables).

[5] An equivalent antecedent is $\vdash \tau \lesssim_i \tau'$; it is equivalent because $\tau' = \Delta_i(\tau')$.

The Preservation Lemma now expresses that evaluation extends the store consistently. Together with a Store Weakening Lemma, stating that store extensions cannot change the type of an expression, the induction hypothesis is strong enough.

LEMMA 4.1 (STORE WEAKENING). *If Ψ' is an extension of Ψ and $\Psi; \Gamma \vdash e : \tau$, then $\Psi'; \Gamma \vdash e : \tau$.*

LEMMA 4.2 (PRESERVATION). *If $\vdash M : \Psi$, $\Psi; \emptyset \vdash e : \tau$, and $(M, e) \longmapsto (M', e')$, then there exists a Ψ' such that Ψ' extends Ψ, $\vdash M' : \Psi'$, and $\Psi'; \emptyset \vdash e' : \tau$.*

Note that this strengthening of Preservation is precisely what is needed to extend the lambda calculus with state; principals add no complications. As for the proof, the careful use of $\bar{\Delta}_i$ in the static semantics makes it routine. Recall that the proof is by induction on the derivation that $(M, e) \longmapsto (M', e')$. We show only the cases for the rules in Figure 17. The other cases are either simple restatements of the analogous cases from the proof without state or trivial inductive arguments using the Store Weakening Lemma.

PROOF.

[*!1*] By assumption, there must be a derivation of the form

$$\frac{\Psi; \emptyset \vdash v_i : \tau' \quad \bar{\Delta}_i(\tau) = \tau'}{\Psi; \emptyset \vdash \mathbf{ref}_\tau\, v_i : \mathbf{ref}\, \tau} \quad .$$

Letting $M' = M[l^\tau \mapsto v_i]$ and $\Psi' = \Psi[l^\tau \mapsto \tau]$, we have Ψ' extends Ψ. Because we know $\vdash M : \Psi$, to prove $\vdash M' : \Psi'$, it suffices to prove $\Psi'; \emptyset \vdash v_i : \tau'$ and $\bar{\Delta}_i(\tau) = \tau'$. These facts follow from the antecedents of the static derivation and the Store Weakening Lemma. Our final obligation, $\Psi'; \emptyset \vdash l_i^\tau : \mathbf{ref}\, \tau$, follows from the definition of Ψ' and the static semantics.

[*!2*] By assumption, there must be a derivation of the form

$$\frac{\Psi; \emptyset \vdash l_i^\tau : \mathbf{ref}\, \tau \quad \Psi; \emptyset \vdash v_i : \tau' \quad \bar{\Delta}_i(\tau) = \tau'}{\Psi; \emptyset \vdash l_i^\tau := v_i : \tau} \quad .$$

Letting $M' = M[l^\tau \mapsto v_i]$ and $\Psi' = \Psi$ we have that M' and Ψ' are extensions of M and Ψ respectively. We have from the hypothesis of the derivation that $\Psi; \emptyset \vdash v_i : \tau'$, so we have to show only that $\vdash M' : \Psi$. Because we already know $\vdash M : \Psi$, this conclusion follows from $\Psi; \emptyset \vdash v_i : \tau'$ and $\bar{\Delta}_i(\tau) = \tau'$.

[*!3*] and [*!4*] By assumption, there must be a derivation of the form

$$\frac{\Psi; \emptyset \vdash l_i^\tau : \mathbf{ref}\, \tau}{\Psi; \emptyset \vdash !l_i^\tau : \bar{\Delta}_i(\tau).}$$

Letting $M' = M$ and $\Psi' = \Psi$, we need to show only that $\Psi; \emptyset \vdash v_i : \bar{\Delta}_i(\tau)$. There are two cases: $\text{color}(M(l^\tau)) = i$ or not. First assume $\text{color}(M(l^\tau)) = i$. Then $v_i = M(l^\tau)$. It follows from the hypotheses needed to prove $\vdash M : \Psi$ that $\Psi; \emptyset \vdash v_i : \tau'$ and $\bar{\Delta}_i(\tau) = \tau'$ from which the conclusion is immediate. Now assume $M(l^\tau)$ is a different color. Then $v_i = \llcorner M(l^\tau) \lrcorner_j^\tau$. By the argument given for the other case, we know $\Psi; \emptyset \vdash M(l^\tau) : \tau'$ and $\bar{\Delta}_j(\tau) = \tau'$. Hence by [*eq*] we conclude $\tau' \lesssim_j \tau$. Hence [*embed*] applies, and its conclusion is the desired result.

[*!5*] By induction.

[!6] By assumption, there must be a derivation of the form

$$\dfrac{\dfrac{l^\tau \in Dom(\Psi)}{\Psi;\emptyset \vdash l^\tau_j : \mathbf{ref}\,\tau} \quad \vdash \mathbf{ref}\,\tau \lesssim_{\ell i} \mathbf{ref}\,\tau}{\Psi;\emptyset \vdash \llcorner l^\tau_j \lrcorner^{\mathbf{ref}\,\tau}_{\ell} : \bar{\Delta}_i(\mathbf{ref}\,\tau)} \quad .$$

Because $\bar{\Delta}_i(\mathbf{ref}\ \tau) = \mathbf{ref}\ \tau$ and from hypothesis we have $l^\tau \in Dom(\Psi)$, we can derive $\Psi;\emptyset \vdash l^\tau_i : \mathbf{ref}\,\tau$ using [*label*]. Letting $M' = M$ and $\Psi' = \Psi$, we are done. □

The statement of the Progress Lemma remains unchanged other than to use program states instead of expressions. The additions to the proof are entirely straightforward. In the case of $(M, \llcorner \hat{v}_j \lrcorner^\tau_\ell)$, there is one additional case because τ could have the form $\mathbf{ref}\,\tau'$. In this case, Canonical Forms guarantees that $\hat{v}_j$ has the form $l^{\tau'}_j$, so we can take a step using [!6].

Defining erasure from this stateful language with embeddings to the calculus in Figure 16 such that erasure commutes with evaluation is straightforward, so we omit the details. Note that the target of this transformation does not include explicit types on labels; these explicit types have no real dynamic effect.

Extending the Value Abstraction Lemma (to which the Independence of Evaluation Theorem is a corollary) to our stateful calculus is straightforward except for some technical distractions. First, we must use the notion of reachable expressions, which includes expressions in the store, instead of the simpler notion of subterms. Second, the Value Abstraction Lemma uses substitution (such as $\{\hat{v}_j/x_j\}e_i$) to establish a correspondence between two evaluations. To use this technique, we extend the definition of substitution to work on stores in a pointwise fashion. Otherwise the Lemma would not apply if an agent stored $\llcorner \hat{v}_j \lrcorner^\tau_{j\ell}$ in the store and, hence, would not be strong enough to prove the Independence of Evaluation Theorem. Substituting through a store requires that we define a well-formed open store with respect to a context Γ, so that $\Gamma \vdash M : \Psi$ makes sense. The correct definition is the obvious generalization: $\Gamma \vdash M : \Psi$ if for all $l^\tau \in Dom(M)$, $\Psi;\Gamma \vdash M(l^\tau) : \tau'$ and $\bar{\Delta}_i(\tau) = \tau'$, where $M(l^\tau)$ is a possibly-open i-value.

Extending the Host-Provided Values Theorem to our stateful calculus is also straightforward. Essentially, the preservation property needs to be inductively defined to include all terms that are reachable from the current program state without considering the trusted `ho` term. That is, `ho` can maintain private state without invalidating the theorem.

5. POLYMORPHISM AND RECURSIVE TYPES

In this section, we explore the extension of the multiagent language of Section 3 to include recursive types and polymorphism. Both of these mechanisms define type variables: the type $\mu\alpha.\tau(\alpha)$ establishes the equation $\alpha = \tau(\alpha)$. Instantiation of a polymorphic value, $e\!:\!\forall\alpha.\tau$, at type τ' establishes a context in which $\alpha = \tau'$ and e has type τ (that is, $\{\tau'/\alpha\}\tau$).

Because the δ_i maps of the multiagent calculus already provide a way of expressing equalities of the form $\alpha = \tau$, we may try to encode recursive or polymorphic types using them. The current multiagent language is not sufficient to encode either of them. The following sections discuss the consequences of adding the necessary mechanisms.

$$\begin{array}{rrcl}
(\text{types}) & \tau & ::= & \ldots \mid \alpha \mid \mu\alpha.\tau \\
(i\text{-terms}) & e_i & ::= & \ldots \mid \mathbf{roll}_{\mu\alpha.\tau}\ e_i \mid \mathbf{unroll}\ e_i \\
(i\text{-primvals}) & \hat{v}_i & ::= & \ldots \mid \mathbf{roll}_{\mu\alpha.\tau}\ v_i
\end{array}$$

$$[\mu 1]\ \frac{e_i \longmapsto e_i'}{\mathbf{roll}_{\mu\alpha.\tau}\ e_i \longmapsto \mathbf{roll}_{\mu\alpha.\tau}\ e_i'} \qquad [\mu 2]\ \frac{e_i \longmapsto e_i'}{\mathbf{unroll}\ e_i \longmapsto \mathbf{unroll}\ e_i'}$$

$$[\mu 3]\ \ \mathbf{unroll}\ \mathbf{roll}_{\mu\alpha.\tau}\ v_i \ \longmapsto\ v_i$$

$$[\mu 4]\ \ \lceil \mathbf{roll}_{\mu\alpha.\tau}\ \hat{v}_j \rceil_{j\ell}^{\mu\alpha.\tau'} \ \longmapsto\ \mathbf{roll}_{\mu\alpha.\tau'}\ \lceil \hat{v}_j \rceil_{j\ell}^{\{\mu\alpha.\tau'/\alpha\}\tau'} \quad (\mu\alpha.\tau' = \bar{\Delta}_i(\mu\alpha.\tau'))$$

$$[roll]\ \frac{\Gamma \vdash e_i : \{\mu\alpha.\tau/\alpha\}\tau}{\Gamma \vdash \mathbf{roll}_{\mu\alpha.\tau}\ e_i : \mu\alpha.\tau} \qquad [unroll]\ \frac{\Gamma \vdash e_i : \mu\alpha.\tau}{\Gamma \vdash \mathbf{unroll}\ e_i : \{\mu\alpha.\tau/\alpha\}\tau}$$

Fig. 19. Recursive types as an orthogonal extension.

5.1 Recursive Types

There are two general approaches for adding recursive types to the multiagent language. The first way is straightforward, and it interacts well with the previous results about the language. The idea is to treat μ-bound type variables as separate entities from the type variables that are used for abstraction (they could be drawn from the same syntactic class, but separating them makes for cleaner presentation). Figure 19 contains the necessary additions to the language.

The new types include recursive type variables, ranged over by α, and the recursive types themselves, $\mu\alpha.\tau$. Terms **roll** and **unroll** mediate the isomorphism between $\mu\alpha.\tau$ and $\{\mu\alpha.\tau/\alpha\}\tau$. If v_i is an i-value, then $\mathbf{roll}_{\mu\alpha.\tau}\ v_i$ is an i-primval. The typing judgments for the new forms are standard.[6] The operational rules [$\mu 1$], [$\mu 2$], and [$\mu 3$] are also standard—they allow progress under a **roll** or **unroll** and establish that composing **unroll** with **roll** is the identity.

The new rule, [$\mu 4$], propagates the embedding on a **roll** expression that is exported at a concrete type. It is similar to rule [*9*] used for embedded functions in that the outer agent may have different information about the nature of the recursive type. For instance, the outer agent may know that the value is of type $\mu\alpha.1+\mathsf{t}\times\alpha$, the type of t-lists, whereas the inner agent, for whom $\delta_j(\mathsf{t}) = int$ views the value as an *int*-list.[7] The side condition on [$\mu 4$] ensures that the dynamic semantics are deterministic—rule [*7*] applies to a disjoint collection of terms.

The only remaining change needed to establish the soundness of this system is to extend the definition of Δ_i in terms of δ_i.

$$\begin{array}{rcl}
\Delta_i(\mu\alpha.\tau) & = & \mu\alpha.\Delta_i(\tau) \\
\Delta_i(\alpha) & = & \alpha
\end{array}$$

[6] As in the case for λ-abstractions, types annotating **roll** terms are required to be the most concrete possible, but we do not need to apply Δ_i because the concreteness of the type in the antecedent implies the concreteness of the type on the **roll**.

[7] We did not formally add product and sum types to our framework, but doing so is straightforward. For example, $\lceil \mathbf{inleft}_{\tau^0+\tau^1}\, e_j \rceil_\ell^{\tau^2+\tau^3} \longmapsto \mathbf{inleft}_{\tau^2+\tau^3} \lceil e_j \rceil_\ell^{\tau^2}$ when τ^2 and τ^3 are most concrete.

Because the μ-bound type variables are syntactically distinct from the abstraction type variables, it makes no sense for α to be in the domain of δ_i. To avoid problems with capture, we prohibit free occurrences of α in the range of δ_i.

Having made these additions to the language, it is straightforward to establish type soundness. To establish Preservation for the case of [*μ4*], we need a type-relations lemma that is analogous to the Arrow Lemma used in the case of [*9*]:

LEMMA 5.1 (μ TYPE RELATIONS). *If* $\vdash \mu\alpha.\tau \lesssim_\ell \mu\alpha.\tau'$, *then* $\vdash \{\mu\alpha.\tau/\alpha\}\tau \lesssim_\ell \{\mu\alpha.\tau'/\alpha\}\tau'$

The most interesting case of the proof of preservation is when the last rule of the dynamic derivation is [*μ4*]:

PROOF. By assumption, we have a derivation of the form

$$\frac{\dfrac{\emptyset \vdash e_j : \{\mu\alpha.\tau/\alpha\}\tau}{\emptyset \vdash \mathbf{roll}_{\mu\alpha.\tau}\ e_j : \mu\alpha.\tau} \quad \vdash \mu\alpha.\tau \lesssim_{j\ell i} \mu\alpha.\tau'}{\emptyset \vdash \llcorner \mathbf{roll}_{\mu\alpha.\tau}\ e_j \lrcorner_{j\ell}^{\mu\alpha.\tau'} : \bar{\Delta}_i(\mu\alpha.\tau')} \quad .$$

Using the right antecedent and Lemma 5.1, we conclude that $\vdash \{\mu\alpha.\tau/\alpha\}\tau \lesssim_{j\ell i} \{\mu\alpha.\tau'/\alpha\}\tau'$. So using the top antecedent, we can derive

$$\frac{\emptyset \vdash e_j : \{\mu\alpha.\tau/\alpha\}\tau \quad \vdash \{\mu\alpha.\tau/\alpha\}\tau \lesssim_{j\ell i} \{\mu\alpha.\tau'/\alpha\}\tau'}{\emptyset \vdash \llcorner e_j \lrcorner_{j\ell}^{\{\mu\alpha.\tau'/\alpha\}\tau'} : \bar{\Delta}_i(\{\mu\alpha.\tau'/\alpha\}\tau')} \quad .$$

By assumption, we have the side condition $\bar{\Delta}_i(\mu\alpha.\tau') = \mu\alpha.\tau'$. From this condition, it is easy to show that $\bar{\Delta}_i(\{\mu\alpha.\tau'/\alpha\}\tau') = \{\bar{\Delta}_i(\mu\alpha.\tau')/\alpha\}\bar{\Delta}_i(\tau') = \{\mu\alpha.\tau'/\alpha\}\tau'$. So we can derive

$$\frac{\emptyset \vdash \llcorner e_j \lrcorner_{j\ell}^{\{\mu\alpha.\tau'/\alpha\}\tau'} : \{\mu\alpha.\tau'/\alpha\}\tau'}{\emptyset \vdash \mathbf{roll}_{\mu\alpha.\tau'}\ \llcorner e_j \lrcorner_{j\ell}^{\{\mu\alpha.\tau'/\alpha\}\tau'} : \mu\alpha.\tau'.}$$

□

The Progress Lemma is routine; it uses an extension of the Canonical Forms Lemma that asserts that i-values of type $\mu\alpha.\tau$ have the form $\mathbf{roll}_{\mu\alpha.\tau}\ v_i$.

This way of dealing with recursive types has the advantage of simplicity: because the presence of recursive types is orthogonal to the kind of type abstraction allowed by embeddings, all of the proofs of safety properties in Section 3 require minimal changes to account for the new constructs.

Another approach to incorporating recursive types into this system requires us to change the notion of compatibility of the Δ_i maps to account for cycles. Given a recursive type $\mu\alpha.\tau$, observe that the **roll**ed values behave essentially abstractly. There is only one destructor for the term $\mathbf{roll}_{\mu\alpha.\tau}\ v_i$, namely, **unroll**. This observation suggests that we can encode recursive types like so: Let r and u be fresh type variables. Create a special agent, r, whose sole function is to provide the **roll** and **unroll** operations for a recursive type. From the type perspective, r knows that $\mathsf{r} = \{\mathsf{u}/\alpha\}\tau$, i.e., $\delta_r(\mathsf{r}) = \{\mathsf{u}/\alpha\}\tau$, and δ_r is otherwise undefined. For any client agent, i, who wishes to use values of type $\mu\alpha.\tau$, let $\delta_i(\mathsf{u}) = \mathsf{r}$. The terms encoding $\mathbf{roll}_{\mu\alpha.\tau}\ x$ and $\mathbf{unroll}\ x$ are

$$\begin{aligned}\mathbf{roll}_{\mu\alpha.\tau} &\equiv \llcorner\lambda x_r : \{\mathsf{u}/\alpha\}\tau.\ x_r\lrcorner_r^{\{\mathsf{u}/\alpha\}\tau\to\mathsf{r}}\\ \mathbf{unroll} &\equiv \llcorner\lambda x_r : \{\mathsf{u}/\alpha\}\tau.\ x_r\lrcorner_r^{\mathsf{r}\to\{\mathsf{u}/\alpha\}\tau}\end{aligned}$$

The type variable r acts like the type $\mu\alpha.\tau$. As expected, the agent r provides identity functions that outwardly appear to clients as having the types appropriate to **roll** and **unroll**: $\{\mathsf{r}/\alpha\}\tau \to \mathsf{r}$ and $\mathsf{r} \to \{\mathsf{r}/\alpha\}\tau$, respectively.

Although this approach is initially appealing, it requires substantial changes to the language. The essential difficulty is that there may not be a most concrete form for a given type variable, due to the possibly cyclic nature of the type information.

Two reasonable alternatives are (1) move to an operational semantics in which types are refined nondeterministically, or (2) require that each agent's type information be acyclic, even if there are cycles globally. Neither of these options is satisfactory. The first makes reasoning about the behavior of programs unduly complicated. The second choice means that giving more type information to an agent can destroy a system's compatibility. Both suffer from the problem that erasure (to a typed language) is harder to establish.

There may be merit in further considering this approach, particularly if the agents are module-level constructs and if the concern is mutually recursive data types that cross module boundaries. For a core language, however, the simplicity of the first approach and its compatibility with our previous proofs make it more valuable.

5.2 Polymorphism

The primary difference between polymorphism and the type abstraction already present in the multiagent calculus is one of scope: the abstract types present in the multiagent calculus are global and static, whereas a polymorphic instantiation is local to part of the program. That is, the application $(\Lambda\alpha.\ e)\ \tau$ introduces the type equality $\alpha = \tau$ within the scope of the expression e. A second difference between type abstraction and polymorphism is that, in the latter, the same type variable may be instantiated at different types, for example,

$$\lambda x : \forall\alpha.\alpha \to \alpha.\ (\ldots (x\ [int]) \ldots (x\ [bool]) \ldots).$$

It appears that $\alpha = int$ holds for part of the term whereas $\alpha = bool$ holds for a different part of the term.

This second difference is superficial, however. Because the α in x's type is bound in $\forall\alpha.\alpha \to \alpha$, we are free to rename it for each occurrence of x in the body. We can think of the term $(x\ [int])$ as establishing the equation $\alpha = int$ whereas in the second term we choose the type of x to be (the equivalent type) $\forall\beta.\beta \to \beta$, and thus $(x\ [bool])$ defines $\beta = bool$. In this way, each type application can establish the concrete form of a distinct type variable.

The true distinction, then, between our type-abstraction mechanism and polymorphism is local scoping. As with our first approach to recursive types, we could include both features separately. Instead, we investigate how to extend our framework to unify type abstraction and polymorphism via a single mechanism. Bridging the gap between the two features amounts to allowing the definitions of the δ_i maps to change during evaluation. We use the notation $\{\Delta\}$ to mean a set of type maps $\{\Delta_1, \ldots, \Delta_n\}$; it captures the type knowledge each agent has at a particular point

$$
\begin{array}{rlll}
\text{(types)} & \tau & ::= \ \ldots & \mid \ \forall\alpha.\tau \\
\text{(i-terms)} & e_i & ::= \ \ldots & \mid \ \Lambda\alpha.\, e_i \ \mid \ e_i\ [\tau] \\
\text{(i-primvals)} & \hat{v}_i & ::= \ \ldots & \mid \ \Lambda\alpha.\, e_i
\end{array}
$$

$$
Dom\{\Delta\} = \bigcup_i Dom(\delta_i)
$$

$$
[\forall 1] \quad \langle\{\Delta\}, (\Lambda\alpha.\, e_i)\ [\tau]\rangle \ \longmapsto\ \langle\{\Delta\} \uplus_i \{\alpha = \tau\}, \{\tau/\alpha\}_i e_i\rangle
$$

$$
[\forall 2] \quad \langle\{\Delta\}, \ulcorner\Lambda\alpha.\, e_j\urcorner_\ell^{\forall\alpha.\tau}\rangle \ \longmapsto\ \langle\{\Delta\}, \Lambda\alpha.\, \ulcorner e_j\urcorner_\ell^{\tau}\rangle
$$

$$
[\forall intro] \quad \frac{\Theta, \alpha; \{\Delta\}; \Gamma \vdash e_i : \tau}{\Theta; \{\Delta\}; \Gamma \vdash \Lambda\alpha.\, e_i : \forall\alpha.\tau} \ (\alpha \notin \Theta \cup Dom\{\Delta\})
$$

$$
[\forall elim] \quad \frac{\Theta; \{\Delta\}; \Gamma \vdash e_i : \forall\alpha.\tau \quad \Delta_i(\tau') = \tau'}{\Theta; \{\Delta\}; \Gamma \vdash e_i\ [\tau'] : \{\tau'/\alpha\}\tau} \ (\alpha \notin Dom\{\Delta\})
$$

Fig. 20. Polymorphism.

in the program evaluation. We can then integrate traditional polymorphic terms by extending our notion of evaluation from $e_i \longmapsto e_i'$ to $\langle\{\Delta\}, e_i\rangle \longmapsto \langle\{\Delta'\}, e_i'\rangle$. This approach is similar to the allocation-based, explicit type-passing semantics for polymorphism found in the dissertation of Morrisett [1995].

Figure 20 contains the necessary adjustments to the language. Note that we have only one form of type variable. In keeping with convention with respect to polymorphism, we allow α and β to range over type variables in addition to t, s, and u. Types are extended with the form $\forall\alpha.\tau$. We add terms of the form $\Lambda\alpha.\, e_i$, in which α is abstract in e_i.

Instantiation of a type variable is represented by $e_i\ [\tau]$. Operationally (see rule $[\forall 1]$), this application corresponds to extending δ_i to include $\alpha = \tau$. We use the notation $\Delta_i \uplus \{\alpha = \tau\}$ to mean the mapping from types to types obtained from $\delta_i[\alpha \mapsto \tau]$. The notation $\{\Delta\} \uplus_i \{\alpha = \tau\}$ represents

$$
\{\Delta_1, \ldots, \Delta_{i-1}, \Delta_i \uplus \{\alpha = \tau\}, \Delta_{i+1}, \ldots, \Delta_n\}.
$$

This extension is valid only if it preserves the compatibility of the Δ maps. Letting $Dom\{\Delta\}$ mean $\bigcup_i Dom(\delta_i)$, it is easy to show that if $\{\Delta\}$ is compatible and $\alpha \notin Dom\{\Delta\}$, then $\{\Delta\} \uplus_i \{\alpha = \tau\}$ is compatible. As a notational convenience, whenever we use $\{\Delta\} \uplus_i \{\alpha = \tau\}$, we implicitly assume that $\alpha \notin Dom\{\Delta\}$.

We need to specify the behavior of Δ_i on types of the form $\forall\alpha.\tau$. Assuming $\alpha \notin Dom(\delta_i)$, which is always possible via alpha-conversion of $\forall\alpha.\tau$, we have

$$
\Delta_i(\forall\alpha.\tau) = \forall\alpha.\Delta_i(\tau).
$$

The original multiagent calculus maintains the invariant that any types appearing as part of the syntax of an i-term are most concrete from agent i's perspective. This invariant is enforced in the typing rules; for example, the $[abs]$ rule requires that $\Delta_i(\tau') = \tau'$. Unfortunately, this invariant is harder to maintain now that an agent may "learn" information about a type variable at runtime. To do so, we introduce a special substitution operator, $\{\tau/\alpha\}_i$, to perform the substitution of τ for α only in terms colored i, including i-subterms of any j-colored subexpressions. We can extend the operator to contexts in a pointwise manner because variables in the context have a color.

We can now understand the operational behavior of the new constructs. Polymorphic expressions are values. Rule [$\forall$*1*] performs type instantiation. Agent i's type map is extended with the new binding for α, and we substitute τ for α in the i-colored terms of the body. Most importantly, no other agents knows α. That way, if the body of the polymorphic expression is a different color, then we will be able to use our techniques to argue that (under sufficient conditions) it evaluates independently of certain values it is passed. Rule [$\forall$*2*] describes the transition step for an embedded polymorphic value being exported as such. In this case, the type maps do not change; the outer embedding is simply moved inside the Λ.

The static semantics are derived from the regular polymorphic lambda calculus with a few twists to account for our different style of tracking type information. First, because $\{\Delta\}$ changes during the course of evaluation, we must parameterize our typing judgments by the current definition of $\{\Delta\}$. We also need to track the lexical scoping of Λ-bound type variables. We do so by adding a set, Θ, of type variables currently in scope. The new form of typing judgments is therefore $\Theta; \{\Delta\}; \Gamma \vdash e_i : \tau$.

With these additions, the static rules of the original multiagent calculus carry over to the new setting, with the understanding that occurrences of Δ_i in those judgments now refer to the Δ_i found in the $\{\Delta\}$ parameter to the rule. (Θ is unused by the new version of the old rules.) The typing rules for the new constructs are [$\forall$*intro*] and [$\forall$*elim*]. The former lets us conclude that a polymorphic term has type $\forall\alpha.\tau$ if, when we add α to the variables in scope, the body has type τ. The side condition ensures that α is distinct from all other variables in scope and distinct from those variables that are defined in any δ map. (We can always satisfy this condition by suitable alpha conversion.) Note that this condition means $\Delta_i(\alpha) = \alpha$ for any agent i, and hence τ may mention α.

The rule [$\forall$*elim*] shows how to type polymorphic instantiation. Because agent i is performing the instantiation, we require that the instantiation type is most concrete from the perspective of agent i. The last antecedent requires that such an update preserves compatibility.

One might expect that the static scoping of type variables regulated by Θ and the dynamic information encapsulated in $\{\Delta\}$ are closely related. The following lemma, which we need to prove preservation, establishes the relation.

LEMMA 5.2. *Suppose* $\Theta, \alpha; \{\Delta\}; \Gamma \vdash e_j : \tau'$ *and* $\alpha \notin Dom\{\Delta\}$. *Let* $\{\Delta'\} = \{\Delta\} \uplus_i \{\alpha = \tau\}$. *Then*

(i) If $i = j$ *then* $\Theta; \{\Delta'\}; \{\tau/\alpha\}_i\Gamma \vdash \{\tau/\alpha\}_i e_j : \{\tau/\alpha\}\tau'$

(ii) If $i \neq j$ *then* $\Theta; \{\Delta'\}; \{\tau/\alpha\}_i\Gamma \vdash \{\tau/\alpha\}_i e_j : \tau'$.

PROOF SKETCH. By simultaneous induction of parts (i) and (ii) on the derivation that $\Theta, \alpha; \{\Delta\}; \Gamma \vdash e_j : \tau'$. $\square$

As usual, when we add a new type constructor to the language, we need a Type Relations Lemma to relate subcomponents of related types. In this case, we have the following:

Lemma 5.3 ($\forall$ Type Relations). *If* $\{\Delta\} \vdash \forall\alpha.\tau \lesssim_\ell \forall\alpha.\tau'$*, then* $\{\Delta\} \vdash \tau \lesssim_\ell \tau'$*.*

We alter the statement of preservation to account for the difference in program configurations. As usual for an allocation-style semantics, we must show that the program does not lose type information during the course of evaluation. Thus, we make the following definition:

Definition 5.4. Let $\{\Delta\} \leq \{\Delta'\}$ mean $\{\Delta'\} = \{\Delta\} \uplus_i \{\alpha = \tau\}$ for some agent i, type τ, and type variable α such that $\alpha \notin Dom\{\Delta\}$. Let $\leq^*$ be the reflexive, transitive closure of $\leq$. We say that $\{\Delta'\}$ *refines* $\{\Delta\}$ whenever $\{\Delta\} \leq^* \{\Delta'\}$.

Lemma 5.5 (Preservation). *If* $\emptyset; \{\Delta\}; \emptyset \vdash e_i : \tau$ *and* $\langle\{\Delta\}, e_i\rangle \longmapsto \langle\{\Delta'\}, e_i'\rangle$*, then* $\{\Delta'\}; \emptyset \vdash e_i' : \tau$*. Furthermore,* $\{\Delta'\}$ *is compatible and refines* $\{\Delta\}$*.*

Proof. The cases for terms not involving the new constructs are similar to those cases presented earlier, so we omit them. The new cases are:

[$\forall$*1*] By assumption, there is a derivation of the form

$$\dfrac{\dfrac{\alpha; \{\Delta\}; \emptyset \vdash e_i : \tau''}{\emptyset; \{\Delta\}; \emptyset \vdash \Lambda\alpha.\, e_i : \forall\alpha.\tau''} \quad \Delta_i(\tau) = \tau}{\emptyset; \{\Delta\}; \emptyset \vdash (\Lambda\alpha.\, e_i)\; [\tau] : \{\tau/\alpha\}\tau''}\; (\alpha \notin Dom\{\Delta\}).$$

Let $\{\Delta'\} = \{\Delta\} \uplus_i \{\alpha = \tau\}$ and $\Delta_i' = \Delta_i \uplus_i \{\alpha = \tau\}$. We can apply Lemma 5.2 to the antecedent to obtain $\emptyset; \{\Delta'\}; \emptyset \vdash \{\tau/\alpha\}_i e_i : \{\tau/\alpha\}\tau''$. Clearly $\{\Delta'\}$ is compatible and refines $\{\Delta\}$.

[$\forall$*2*] By assumption, there is a derivation of the form

$$\dfrac{\dfrac{\alpha; \{\Delta\}; \emptyset \vdash e_j : \tau^1}{\emptyset; \{\Delta\}; \emptyset \vdash \Lambda\alpha.\, e_j : \forall\alpha.\tau^1} \quad \{\Delta\} \vdash \forall\alpha.\tau^1 \lesssim_{j\ell i} \forall\alpha.\tau}{\emptyset; \{\Delta\}; \emptyset \vdash \llcorner \Lambda\alpha.\, e_j \lrcorner_{j\ell}^{\forall\alpha.\tau} : \bar{\Delta}_i(\forall\alpha.\tau)}\;.$$

The side conditions ensure that $\alpha \notin Dom\{\Delta\}$, so we are able to conclude that $\bar{\Delta}_i(\forall\alpha.\tau) = \forall\alpha.\bar{\Delta}_i(\tau)$. We need to construct a derivation for

$$\dfrac{\dfrac{\alpha; \{\Delta\}; \emptyset \vdash e_j : \tau^1 \quad \{\Delta\} \vdash \tau^1 \lesssim_{j\ell i} \tau}{\alpha; \{\Delta\}; \emptyset \vdash \llcorner e_j \lrcorner_{j\ell}^{\tau} : \bar{\Delta}_i(\tau)}}{\emptyset; \{\Delta\}; \emptyset \vdash \Lambda\alpha.\, \llcorner e_j \lrcorner_{j\ell}^{\tau} : \forall\alpha.\bar{\Delta}_i(\tau)}\;.$$

We have the first antecedent from the original typing derivation. The Type Relations Lemma (5.3) says that $\{\Delta\} \vdash \forall\alpha.\tau^1 \lesssim_{j\ell i} \forall\alpha.\tau$ implies $\{\Delta\} \vdash \tau^1 \lesssim_{j\ell i} \tau$. The side condition on [$\forall$*intro*] has already been observed above. Because $\{\Delta\}$ is compatible and refines itself, we are done. $\square$

Because the $\{\Delta\}$ context changes during evaluation, and the definition of when $\llcorner \hat{v}_j \lrcorner_{j\ell}^{t}$ is an i-value depends on whether $\Delta_i(t) = t$, the notion of value is also dynamic. We write $\{\Delta\} \vdash e_i : \text{Value}$ when either e_i is an i-primval or $e_i = \llcorner \hat{v}_j \lrcorner_{\ell}^{t}$ and $t \notin Dom\{\Delta\}$.

Progress, proved as usual by induction on the structure of e_i, then becomes:

Lemma 5.6 (Progress). *If* $\emptyset; \{\Delta\}; \emptyset \vdash e_i : \tau$ *then either* $\{\Delta\} \vdash e_i : \text{Value}$ *or there exists an* e_i' *and* $\{\Delta'\}$ *such that* $\langle\{\Delta\}, e_i\rangle \longmapsto \langle\{\Delta'\}, e_i'\rangle$

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau \to \tau \mid \forall\alpha.\tau \\ e & ::= & x \mid \lambda x{:}\tau.\, e \mid (e\; e) \mid \Lambda\alpha.\, e \mid (e\,[\tau]) \end{array}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma[x{:}\tau] \vdash e : \tau'}{\Gamma \vdash \lambda x{:}\tau.\, e : \tau \to \tau'}\;(x \notin Dom(\Gamma)) \qquad \frac{\Gamma \vdash e : \tau' \to \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash (e\; e') : \tau}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.\, e : \forall\alpha.\tau}\;(\alpha \notin FTV(\Gamma)) \qquad \frac{\Gamma \vdash e : \forall\alpha.\tau'}{\Gamma \vdash (e\,[\tau]) : \{\tau/\alpha\}\tau'}$$

Fig. 21. Polymorphic lambda calculus.

5.3 Translation from the Polymorphic Lambda Calculus

The net effect of our design decisions is that we can faithfully embed the polymorphic lambda calculus (shown in Figure 21) into this new language by simply decorating the term with a single color. In this case, the operational semantics degenerates to those of the regular polymorphic lambda calculus; in particular, type application still substitutes a type for a type variable. Using such a naive translation, our system will provide no more opportunity to use syntactic proofs of type abstraction than before.

Instead, we show that a smarter translation, which essentially colors the code in such a way as to make agents with different type information explicit, allows us to recover some type-abstraction proofs in a syntactic manner.

We formulate the translation as a type-directed transformation $\mathcal{C}[\![\Gamma \vdash e : \tau]\!]$. The translation takes two additional parameters: i, an agent indicating the desired color of the result term, and γ, a map from source variables to target variables (including color information). We maintain the invariant that, when compiling the judgment $\mathcal{C}[\![\Gamma \vdash e : \tau]\!]i\gamma$, we have $Dom(\gamma) = Dom(\Gamma)$. The result of the translation is an i-term.

The translation is given in Figure 22. Most of the rules are straightforward—they recursively translate the subterms of the expression and combine the results into the corresponding construct. The two interesting cases are [*Tvar*] and [*Ttypeabs*].

The rule for translating the variable x simply looks up x in the map γ. If the resulting variable has the same color as the desired color of the term, no further action is needed. Otherwise, the translation produces an embedding of the variable to yield a term of the appropriate color. Because the type $\Gamma(x)$ may mention type variables that are currently in scope, the embedding may be abstract.

The idea for translating a polymorphic term is to "spawn" a new agent with which to color the body of the type abstraction. By recursively translating the body to a term of this new color, [*Ttypeabs*] explicitly marks the boundaries where type information changes.

As examples, we have the following translations:

$$\mathcal{C}[\![\emptyset \vdash \Lambda\alpha.\, \lambda x{:}\alpha.\, x : \forall\alpha.\alpha \to \alpha]\!]i\emptyset = \Lambda\alpha.\, \llcorner \lambda x_j{:}\alpha.\, x_j \lrcorner_j^{\alpha\to\alpha}$$

$$\mathcal{C}[\![\emptyset \vdash \lambda x{:}\mathsf{b}.\, \Lambda\alpha.\, x : \mathsf{b} \to \forall\alpha.\mathsf{b}]\!]i\emptyset = \lambda x_i{:}\mathsf{b}.\, \Lambda\alpha.\, \llcorner\llcorner x_i \lrcorner_i^{\mathsf{b}} \lrcorner_j^{\mathsf{b}}$$

$$[\mathit{Tvar}] \qquad \mathcal{C}[\![\Gamma \vdash x : \Gamma(x)]\!]i\gamma = \begin{cases} \gamma(x) & \text{if } \mathit{color}(\gamma(x)) = i \\ \llcorner\gamma(x)\lrcorner_j^{\Gamma(x)} & \text{if } \mathit{color}(\gamma(x)) = j \neq i \end{cases}$$

$$[\mathit{Tabs}] \qquad \frac{\mathcal{C}[\![\Gamma[x\!:\!\tau] \vdash e : \tau']\!]i(\gamma[x \mapsto x_i]) = e_i}{\mathcal{C}[\![\Gamma \vdash \lambda x\!:\!\tau.\ e : \tau \to \tau']\!]i\gamma = \lambda x_i\!:\!\tau.\ e_i}$$

$$[\mathit{Tapp}] \qquad \frac{\mathcal{C}[\![\Gamma \vdash e : \tau' \to \tau]\!]i\gamma = e_i \quad \mathcal{C}[\![\Gamma \vdash e' : \tau']\!]i\gamma = e'_i}{\mathcal{C}[\![\Gamma \vdash (e\ e') : \tau]\!]i\gamma = (e_i\ e'_i)}$$

$$[\mathit{Ttypeabs}] \qquad \frac{\mathcal{C}[\![\Gamma \vdash e : \tau]\!]j\gamma = e_j}{\mathcal{C}[\![\Gamma \vdash \Lambda\alpha.\ e : \forall\alpha.\tau]\!]i\gamma = \Lambda\alpha.\ \llcorner e_j\lrcorner_j^{\tau}}\ (j \text{ fresh})$$

$$[\mathit{Ttypeapp}] \qquad \frac{\mathcal{C}[\![\Gamma \vdash e : \forall\alpha.\tau]\!]i\gamma = e_i}{\mathcal{C}[\![\Gamma \vdash (e\ [\tau']) : \{\tau'/\alpha\}\tau]\!]i\gamma = (e_i\ [\tau'])}$$

Fig. 22. Translation of the polymorphic lambda calculus.

The translation is type-preserving, as formalized in the following lemma, where $\gamma\Gamma$ is the pointwise application of γ to the variables in Γ.

Lemma 5.7. *Suppose $\Gamma \vdash e : \tau$ in the polymorphic lambda calculus and γ is a map from source variables to target variables where $Dom(\gamma) = Dom(\Gamma)$. Suppose further that $\mathcal{C}[\![\Gamma \vdash e : \tau]\!]i\gamma = e_i$. Let $\{\Delta\} = \{\Delta_k \mid k \in \text{Agents}(e_i), \delta_k = \emptyset\}$, and let $\Theta = FTV(\Gamma)$, the free type variables appearing in Γ. Then $\Theta; \{\Delta\}; \gamma\Gamma \vdash e_i : \tau$.*

Proof Sketch. By induction on the derivation that $\Gamma \vdash e : \tau$ in the source language. The base case is handled by the definition of γ. It uses the fact that for any $\{\Delta\}$ and nonempty ℓ, $\{\Delta\} \vdash \tau \lesssim_\ell \tau$. The same idea is used to show that the embedding introduced in the translation of a polymorphic term is well-typed. To show that a translated type application is well-formed, we need to show that $\alpha \notin Dom\{\Delta\}$ and that $\Delta_i(\tau') = \tau'$, but these follow from the fact that each δ_k is the empty map. The remainder of the cases follow by straightforward induction. □

We can erase this extended multiagent calculus (excluding **fix**) to the polymorphic lambda calculus by extending our definition of erasure to include the new terms. Also, the definition of Φ (the composite type information) depends on the current $\{\Delta\}$, so we treat $\{\Delta\}$as an input to *erase*.

$$\begin{aligned} \mathit{erase}(\{\Delta\}, \Lambda\alpha.\ \llcorner e_j\lrcorner_{j\ell}^{\tau}) &= \Lambda\alpha.\ \mathit{erase}(\{\Delta\}, e_j) \\ \mathit{erase}(\{\Delta\}, e_i\ [\tau]) &= \mathit{erase}(\{\Delta\}, e_i)\ [\bar{\Phi}(\tau)] \end{aligned}$$

Using the same techniques as before, we prove the following lemma:

Lemma 5.8 (Erasure). *If e_i is well-typed and $\langle\{\Delta\}, e_i\rangle \longmapsto^* \langle\{\Delta'\}, e'_i\rangle$, then $\mathit{erase}(\{\Delta\}, e_i) \longmapsto^* \mathit{erase}(\{\Delta'\}, e'_i)$.*

This lemma lets us establish a correspondence between the polymorphic lambda calculus and its translation into our multiagent setting.

Lemma 5.9 (Translation). *If $\emptyset \vdash e : \tau$ in the polymorphic lambda calculus, then $\mathit{erase}(\mathcal{C}[\![\emptyset \vdash e : \tau]\!]i\emptyset) = e$.*

PROOF SKETCH. By induction on the derivation that e is well-typed. The invariant must be strengthened to include open terms. $\square$

Putting the previous two lemmas together yields the following theorem, which establishes the correctness of our translation:

THEOREM 5.10 (CORRECTNESS OF TRANSLATION). *Suppose* $\emptyset \vdash e : \tau$ *is a term of the polymorphic lambda calculus,* $e \longmapsto^* v$*, and* $\mathcal{C}[\![\emptyset \vdash e : \tau]\!] i \emptyset = e_i$*. Furthermore, let* $\{\Delta\} = \{\Delta_k \mid k \in \mathrm{Agents}(e_i), \delta_k = \emptyset\}$*. Then there exists a* v_i *such that* $\langle \{\Delta\}, e_i \rangle \longmapsto^* \langle \{\Delta'\}, v_i \rangle$ *and* $erase(\{\Delta'\}, v_i) = v$.

Finally, if we return to the file-handle example from the introduction, letting $\tau_c = string \to \mathsf{fh}$, we see that

$$\begin{gathered}\mathcal{C}[\![(\Lambda \mathsf{fh}.\ \lambda \mathtt{host}\!:\!\tau_c.\ client\_code)\ int\ host\_code]\!] h \emptyset \\ = \\ (\Lambda \mathsf{fh}.\ \lceil \lambda \mathtt{host}_c\!:\!\tau_c.\ client\_code_c \rceil_c^{\tau_c \to \tau})\ int\ host\_code_h.\end{gathered}$$

If we assume that $host\_code_h$ is a value, the Canonical Forms lemma tells us that it is a function, $\lambda x_h : string.\ host$. Let τ_h be $string \to int$, the host's view of τ_c. Let $\{\Delta\}$ be $\{\delta_h = \emptyset, \delta_c = \emptyset\}$ and $\{\Delta'\}$ be $\{\delta_h = [\mathsf{fh} \mapsto int], \delta_c = \emptyset\}$. Then the first several steps of the operational semantics are

$$\begin{aligned} & \langle \{\Delta\}, (\Lambda \mathsf{fh}.\ \lceil \lambda \mathtt{host}_c\!:\!\tau_c.\ client\_code_c \rceil_c^{\tau_c \to \tau})\ int\ host\_code_h \rangle \\ \longmapsto\ & \langle \{\Delta'\}, \lceil \lambda \mathtt{host}_c\!:\!\tau_c.\ client\_code_c \rceil_c^{\tau_h \to \tau}\ host\_code_h \rangle \\ \longmapsto\ & \langle \{\Delta'\}, (\lambda \mathtt{host}_h\!:\!\tau_h.\ \lceil \{\lceil \mathtt{host}_h \rceil_h^{\tau_c} / \mathtt{host}_c\} client\_code_c \rceil_c^{\tau})\ host\_code_h \rangle \\ \longmapsto\ & \langle \{\Delta'\}, \lceil \{\lceil host\_code_h \rceil_h^{\tau_c} / \mathtt{host}_c\} client\_code_c \rceil_c^{\tau} \rangle. \end{aligned}$$

The evaluation from this point on is via rule [*3*]. We can use the invariant φ presented for the Host-Provided Preservation Lemma: if we assume that *client_code* is host-free, it is easy to establish that $\varphi(client\_code)$. By rule $[H]$, we have $\varphi(\lceil host\_code \rceil_h^{\tau_h})$. Thus, by Host-Provided Substitution we are able to conclude $\varphi(\{\lceil host\_code_h \rceil_h^{\tau_a} / \mathtt{host}_c\} client\_code_c)$. We conclude that any closed value of type fh appearing in the client code during evaluation was obtained through the host interface. (Technically, we should require that the bodies of the client and host do not themselves contain polymorphic terms, because the proof in Section 3 does not account for the new extensions to the language. We expect that this requirement is actually not necessary.)

6. RELATED WORK

There has been much work on representation independence and parametric polymorphism, as pioneered by Strachey [1967] and Reynolds [1983], and more recently by Ma and Reynolds [1992]. Such notions have been incorporated into programming languages such as SML [Milner et al. 1997] and Haskell [Peyton Jones et al. 1999] and studied extensively in Girard's System F [1989]. The connection between type abstraction and existential types has been studied by Mitchell and Plotkin [1988].

Model-theoretic approaches to studying polymorphism have also enjoyed a rich history, although incorporating notions of recursion have proven to be difficult: see MacQueen et al. [1986], the work of Coquand et al. [1987; 1989], and Abadi and Plotkin [1990]. Pitts [1996] has studied relational properties on recursively defined domains and investigated parametricity in extensions of the polymorphic lambda calculus that include fixpoint recursion [Pitts 2000].

Wright and Felleisen [1994] popularized the use of syntactic techniques for proving type safety. We can view our work as extending theirs to prove stronger properties by adding innocuous syntax to a language. Given the right syntactic additions, the resulting proofs follow precisely the subject-reduction form that Wright and Felleisen advocate.

Abadi et al. [1993] have taken a syntactic approach to parametricity by formalizing the logical relations arguments used in such proofs. More recently, Crary [1999] has proposed the use of singleton types as a means of proving parametricity results without resorting to the construction of models.

None of the above work explicitly involves the notion of principal. Our syntactic separation of agents is similar to Nielson and Nielson's two-level lambda calculus [Nielson and Nielson 1992]. 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 [1998], Volpano and Smith's work on type-based security [1997], and the language JFlow [Myers 1999].

Pierce and Sangiorgi [1999] 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. Sewell and Vitek [1999] have also used the idea of coloring in a variant of the pi calculus to prove properties about causal relationships in a security setting.

Perhaps the closest work to ours is Leroy and Rouaix's investigation into the safety properties of typed applets [Leroy and Rouaix 1998]. They use a lambda calculus augmented with state in order to prove theorems similar to our two-agent theorems. They too distinguish between 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.

7. SUMMARY AND CONCLUSIONS

Abstract types are an invaluable tool to software designers. They aid programmers in reasoning about interfaces between different pieces of code.

Despite this utility, it is very hard to prove that the informal reasoning about abstract types, such as, "all file handles are obtained from the `open` system call," is correct. One approach is to encode different principals in the well-understood polymorphic lambda calculus and appeal to results on parametricity. This approach is not always feasible, such as when the language includes state, threads, recursive types, or other expressive features. The syntactic proofs of type-abstraction properties presented in this paper are tedious, but conceptually straightforward. In contrast, building a model is conceptually difficult but can yield elegant proofs.

We have presented a syntactic approach to proving some type-abstraction properties. The main idea is "color" the different principals in the language, each of which has access to different type information. We show how we can track these principals throughout evaluation. This extra syntactic structure gives us a framework in which we can prove properties about where certain values came from (the file-handle example), as well as how abstraction is preserved by evaluation. The arguments take the form of subject-reduction proofs, which show that a desired property is invariant with respect to evaluation, so the hope is that they will scale more easily than model-theoretic techniques. As evidence supporting this hope, we have shown how to extend these results to include mutable references, recursive types, and parametric polymorphism.

There are a number of open questions about our approach. Parametric polymorphism offers the strong notion of representation independence, which can be used to reason about the equivalence of abstract data types. As long as two implementations are equivalent in an appropriate sense, it does not matter which implementation a program uses—they produce equivalent results. We have avoided building such logical relations, but it remains to be seen exactly what formal connections can be drawn between those results and the ones presented here.

We also have not investigated the connections between the multiagent calculus and process-based calculi or object-oriented calculi where the notion of principal (that is, threads or objects) arises naturally. Our method of tracking principals during evaluation should adapt to these domains, but we may also learn more about the multiagent calculus by viewing each agent as a thread and the embedding annotations as points at which synchronization occurs during execution. There is also a similarity between our use (in Section 5.2) of global alpha conversion and the restriction operator, ν, of the pi calculus to generate a "fresh" type variable at runtime.

Another addition to our approach would be an integration of traditional subtyping. That is, we have not considered how we would need to adapt the framework to permit a primitive subtyping relation between (multiple) base types and its standard extension to other type constructors. Our multiagent calculus does allow a principal to reveal partial information about a type. (For example, by letting $t_1 = t_2 \to t_3$, $t_2 = int$, and $t_3 = int$, an agent can export a function of type $int \to int$ at four different types.) Perhaps there is some connection between this sort of partial information and the partial information of knowing that τ is some subtype of τ', but it seems our approach is closer to polymorphism than subtyping.

The kinds of properties we have proven in the multiagent language are limited by the granularity of types. The system does not let us prove that an agent does not misuse file handles, for instance by duplicating them or confusing two different file handles obtained from separate calls to `open`. The reason is that we track embeddings at the granularity of their types—the theorems do not distinguish particular instances of type `fh`. We could imagine tracking such fine-grained details, but the necessary invariants for subject-reduction proofs may become more complex.

Finally, the notion of principal could prove useful for results other than abstraction. For example, we could easily give different operational semantics to different principals and then reason about interlanguage interoperability. Lifting constants and values out of embeddings could require wrapper code or data conversions. Similarly, we could imagine that embeddings represent a form of remote communication. In this case, putting values in embeddings and taking values out would require marshalling and unmarshalling.

REFERENCES

Abadi, M., Cardelli, L., and Curien, P.-L. 1993. Formal parametric polymorphism. In *20th ACM Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 157–167.

Abadi, M. and Plotkin, G. D. 1990. A PER model of polymorphism. In *5th Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, Los Alamitos, CA, 355–365.

Coquand, T., Gunter, C. A., and Winskel, G. 1987. DI-domains as a model of polymorphism. In *Mathematical Foundations of Programming Language Semantics*, M. Main, A. Melton, M. Mislove, and D. A. Schmidt, Eds. Lecture Notes in Computer Science, vol. 298. Springer-Verlag, New York, NY, 344–363.

Coquand, T., Gunter, C. A., and Winskel, G. 1989. Domain theoretic models of polymorphism. *Inf. Comput. 81,* 2 (May), 123–167.

Crary, K. 1999. A simple proof technique for certain parametricity results. In *4th ACM International Conference on Functional Programming*. ACM Press, New York, NY, 82–89.

Girard, J.-Y., Lafont, Y., and Taylor, P. 1989. *Proofs and Types*. Cambridge University Press, Cambridge, UK.

Harper, R. 1994. A simplified account of polymorphic references. *Information Processing Letters 51,* 4 (August), 201–206.

Heintze, N. and Riecke, J. G. 1998. The SLam calculus: programming with secrecy and integrity. In *25th ACM Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 365–377.

Leroy, X. and Rouaix, F. 1998. Security properties of typed applets. In *25th ACM Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 391–403.

Ma, Q. and Reynolds, J. 1992. Types, abstraction, and parametric polymorphism: Part 2. In *Proceedings of the 1991 Mathematical Foundations of Programming Semantics*, S. Brookes, M. Main, A. Melton, M. Mislove, and D. A. Schmidt, Eds. Number 598 in Lecture Notes in Computer Science. Springer-Verlag, New York, NY, 1–40.

MacQueen, D., Plotkin, G. D., and Sethi, R. 1986. An ideal model for recursive polymorphism. *Information and Control 71,* 1/2 (October/November), 95–130.

Milner, R., Tofte, M., Harper, R., and MacQueen, D. 1997. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA.

Mitchell, J. 1991. On the equivalence of data representations. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, V. Lifschitz, Ed. Academic Press, San Diego, CA, 305–330.

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

Morrisett, G. 1995. Compiling with types. Ph.D. thesis, Carnegie Mellon University. Published as CMU Tech Report number CMU-CS-95-226.

Myers, A. C. 1999. JFlow: Practical mostly-static information flow control. In *26th ACM Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 228–241.

Nielson, F. and Nielson, H. R. 1992. *Two-Level Functional Languages*. Number 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK.

Peyton Jones, S., Hughes, J., Augustsson, L., Barton, D., Boutel, B., Fasel, J., Hammond, K., Hinze, R., Hudak, P., Johnsson, T., Jones, M., Launchbury, J., Meijer, E., Peterson, J., Reid, A., Runciman, C., and Wadler, P. 1999. Haskell 98: A non-strict, purely functional language. http://www.haskell.org/onlinereport/.

Pierce, B. C. and Sangiorgi, D. 1999. Behavioral equivalence in the polymorphic pi-calculus. Tech. Rep. MS-CIS-99-10, University of Pennsylvania. Apr. (Summary in 24th ACM Symposium on Principles of Programming Languages).

Pitts, A. M. 1996. Relational properties of domains. *Inf. Comput. 127*, 66–90.

Pitts, A. M. 2000. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science 10*, 1–39.

Reynolds, J. C. 1974. Towards a theory of type structure. In *Programming Symposium.* Lecture Notes in Computer Science, vol. 19. Springer-Verlag, New York, NY, 408–425.

Reynolds, J. C. 1983. Types, abstraction and parametric polymorphism. In *Information Processing 83*, R. E. A. Mason, Ed. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 513–523.

Sewell, P. and Vitek, J. 1999. Secure composition of untrusted code with wrappers and causality types. Work in Progress http://www.cs.purdue.edu/homes/jv/publist.html.

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

Volpano, D. and Smith, G. 1997. A type-based approach to program security. In *TAPSOFT'97, Theory and Practice of Software Development.* Lecture Notes in Computer Science, vol. 1214. Springer-Verlag, New York, NY.

Wright, A. K. and Felleisen, M. 1994. A syntactic approach to type soundness. *Inf. Comput. 115,* 1, 38–94. Preliminary version in Rice Tech. Rep. 91-160.

Zdancewic, S., Grossman, D., and Morrisett, G. 1999. Principals in programming languages: A syntactic proof technique. In *4th ACM International Conference on Functional Programming.* ACM Press, New York, NY, 197–207.

Received February 2000; revised September 2000; accepted November 2000