Scientific Annals in Computer Science vol.18, 2008
"Alexandru Ioan Cuza" University of Iaşi, Romania

A Calculus of Evolving Objects

Mariangiola Dezani-Ciancaglini[1], Paola Giannini[2], Oscar Nierstrasz[3]

Abstract

The demands of developing modern, highly dynamic applications have led to an increasing interest in dynamic programming languages and mechanisms. Not only must applications evolve over time, but the object models themselves may need to be adapted to the requirements of different run-time contexts. Class-based models and prototype-based models, for example, may need to co-exist to meet the demands of dynamically evolving applications. Multi-dimensional dispatch, fine-grained and dynamic software composition, and run-time evolution of behaviour are further examples of diverse mechanisms which may need to co-exist in a dynamically evolving run-time environment. How can we model the semantics of these highly dynamic features, yet still offer some reasonable safety guarantees?

To this end we present an original calculus in which objects can adapt their behaviour at run-time. Both objects and environments are represented by first-class mappings between variables and values. Message sends are dynamically resolved to method calls. Variables may be dynamically bound, making it possible to model a variety of dynamic mechanisms within the same calculus. Despite the highly dynamic nature of the calculus, safety properties are assured by a type assignment system.

[1]Dip. di Informatica, Univ. di Torino, Italy— *www.di.unito.it*
[2]Dip. di Informatica, Univ. del Piemonte Orientale, Italy — *www.di.unipmn.it*
[3]Software Composition Group, University of Bern, Switzerland — *scg.unibe.ch*

1 Introduction

There has been a recent re-emergence of interest in dynamic programming languages [21] and the development of more dynamic features for mainstream languages such as Java. Increasing numbers of applications require the ability for configurations and even system behaviour to evolve at run-time. Furthermore, behaviour may be context-dependent, and may need to adapt to the run-time platform, the end user, service availability, or any number of environmental attributes. To support these highly dynamic applications, programming languages need to support a range of different object models, paradigms and language features.

Multi-dimensional dispatch is one example of a such a feature — instead of dispatching purely on the receiver of a message, the behavior of an object might depend on the sender, or even on contextual information such as the deployment platform, available services, desired quality of service, available versions of components, or even the time of day [14]. Another example is the use of fine-grained components, such as *traits*, to statically or even dynamically extend the behaviour of classes [7]. These and other mechanisms entail the need for specialized lookup mechanisms to adapt the behaviour of objects, even at run-time [24].

It is unclear what the impact of such dynamic features may be on the semantics of programming languages, and on the ability to reason about type safety in the face of dynamic changes. To this end we have developed a stateful calculus of evolving objects in which:

- *Object behaviour is context-dependent* — message-dispatching takes context into account.
- *Objects may change their behaviour at run-time* — message-lookup may be dynamically updated.
- *Dynamic changes are type-safe* — message-not-understood errors are avoided.

Particular innovations of the calculus include:

- The use of *first-class environments* to model both the object states and the environments in which expressions are evaluated.
- The possibility of *binding dynamically variables* by *freezing* expressions containing free variables and *defrosting* them in a runtime environment providing binders for them.

- Distinguishing *message sends* from *method calls* to support object-specific (context-dependent) method lookup.
- A novel *type system* which — in addition to safety properties — assures that variables in an evolving environment are bound to values of fixed types.

The paper is organized as follows. In Section 2 we motivate the calculus through an example. The syntax and the operational semantics of the language are introduced in Sections 3.1 and 3.2. In particular, Section 3.1 introduces the lambda-calculus of environments that is the core functional part of our calculus and in Section 3.2 we add imperative extensible objects in which message send is not identified with method call. In Section 4 we present an overview of the type system with the relevant results. In Section 5 we place our work in context and contrast it to other approaches. We conclude in Section 6 with some remarks on current and future work. The Appendix contains proofs.

2 Motivating Example

In this section we introduce the essential constructs of our calculus with the help of a motivating example.

Suppose we want to model a *Call Center* that answers calls for different clients. When a client calls the Call Center from a known number, then the caller should be directly connected to a dedicated service for that client, for example, to play back recorded calls, or to be connected to the representative for that client. If someone calls from an unknown number, then a default service should be triggered, such as connecting the caller to the switchboard.

In a conventional object-oriented approach, messages are dispatched purely on the basis of the receiver. Each object has its own *methods* for responding to different *messages*. In the Call Center example, the method for responding to a message depends not only on the receiver, but also on the sender. In general, arbitrary contextual information may influence the desired behaviour. For example, depending on the time of day, or the occurrence of a holiday, the Call Center's behaviour might change. To further complicate matters, the behaviour of the Call Center will need to be adapted dynamically as clients come and go.

The operational semantics of object-oriented languages and systems have been extensively studied in the framework of so-called *object calculi*.

Most of these calculi extend the lambda calculus with first-class records used to model objects and classes. Method and field lookup for objects are modeled by looking up fields of records representing classes and objects, possibly following an inheritance chain. The seminal work is by Abadi and Cardelli [1], who developed a series of such object calculi to model various technical aspects such as inheritance, recursion and subtyping.

Other well-known examples of object calculi based on extending the lambda calculus with records include the imperative approach of Flatt *et al.* [10] used to study the addition of mixins to Java-like languages, and the purely functional approach of Igarashi *et al.* in Featherweight Java [15], used to reason about the impact of adding generics to the type system of Java. Researchers more interested in aspects related to concurrency and distribution, on the other hand, have taken process calculi as a starting point [27].

All of these approaches follow an orthodox object-oriented regime in which messages are dispatched on the basis of the receiver.

We propose to extend the conventional approach to object calculi to take context into account when dispatching messages. First, we propose to reify contextual information as *first-class environments*. Although first-class environments have been studied before to model explicit substitutions, they have not been used before to model context-dependent behaviour of objects (see also related work in Section 5). Message lookup can thus take place within a dynamically configured environment. In order to dynamically bind expressions to different environments, however, we need to be able to manipulate expressions containing free variables. We therefore propose a mechanism to *freeze* potentially open expressions and *defrost* them within a given environment. In order to maintain a fine degree of control over the desired semantics of method lookup, we introduce mechanisms that distinguish between *message sending* and *method lookup*. Finally, we ensure that these operations are type-safe.

Our calculus extends a lambda-calculus with explicit substitution and models both execution environments and object fields as sequences of bindings between variables and values, $x_1{=}V_1 \cdots x_n{=}V_n$, denoted by the meta-variable E. Objects are imperative, so each object is associated with an environment that represents its current state.

We represent the Call Center by an object, ι, and the request by sending the message m. Instead of being associated to a field of the object of name m, the method corresponding to m is *dynamically looked up*, with a *lookup*

function for the Call Center, that uses the message name, the information about the sender, and the identity of the sender.

The lookup function needs to manipulate message names as values in order to bind them dynamically to the appropriate environment. We therefore distinguish between two ways to bind the free names when evaluating an expression in an environment, depending on whether they need to be manipulated or not.

- The first is the *sandbox expression* $E;A$, in which the free variables of the expression A must be statically bound to variables defined in the environment E.
- The second is a *conditional expression* $E{\circ}\langle A\rangle{\diamond}B$ to handle the situation where the free variables of A might not all be captured by E. If they are, the conditional reduces to $E;A$ as above, otherwise it reduces to the expression B (analogous to a try/catch block for exceptions).

The conditional expression makes use of the construct $\langle A\rangle$. This *freezes* the expression A, turning it into a closed value even if A contains free variables. In particular, given a (free) message name m, $\langle m\rangle$ is a value, whereas m is not. A frozen expression can be evaluated (defrosted) only in an environment that provides bindings for all its free variables. In case all the free variables of A are defined in E, the expression $E{\circ}\langle A\rangle{\diamond}B$ reduces to $E;A$, thereby *dynamically binding* the free variables of A to the environment E. If E does not define all the free variables of A the evaluation of the expression $E{\circ}\langle A\rangle{\diamond}B$ reduces to B.

Lookup functions can take into account not only the name of the message, but also additional contextual information, such as the identity of the sender. The sender is determined at run-time. In our calculus we provide both a user syntax and a run-time syntax for message sends. The user writes $A\ m(B)$, to send message m to the object denoted by A with B as argument. At run-time the actual message send will be represented by the syntax $E^{\frown}A\ m(B)$, where E provides contextual information concerning the message sender extracted from the execution environment.

Going back to the Call Center example, in case the sender is a client, we can assume that the request arrives from an object whose contextual information contains a binding, $\texttt{client}{=}N$, identifying the sender. If the sender is not a client, no such binding will be present.

We use the notation $*\iota$ to dereference object identifiers, so if ι is the reference to the Call Center, then $*\iota$ denotes the environment associated

with ι. Now we can encode the behaviour of the Call Center in response to a client request as follows:

$$\overline{E}\circ\langle ccClient\ \texttt{client}\ \langle\texttt{request}\rangle\rangle\diamond(*\iota;\texttt{defResp}) \tag{1}$$

where $\overline{E}$ is the environment of the caller, *ccClient* represents the behaviour of the Call Center when the request comes from a client, and `defResp` is the field of the Call Center containing a default behaviour for requests coming from non clients. Let us assume that *ccClient* is a closed expression. Since $\langle\texttt{request}\rangle$ is frozen, hence closed, the *only* free variable of

$$ccClient\ \texttt{client}\ \langle\texttt{request}\rangle \tag{2}$$

is `client`. If we let the binding for `client` in $\overline{E}$ be `client`$=N$, the expression (1) reduces to evaluating (2) in the environment (sandbox) $\overline{E}$ producing:

$$ccClient\ N\ \langle\texttt{request}\rangle.$$

So the Call Center process $\langle\texttt{request}\rangle$ for the client N. If $\overline{E}$ does not have a binding for `client`, then the value of the field `defResp` of the Call Center, is returned. In Fig. 1 we partially define a Call Center object such that a

$$\begin{array}{rcl}
\texttt{lkp} & = & \lambda w.\lambda s.\lambda m.w\circ\langle ccClient\ \texttt{client}\ m\rangle\diamond(*s;\texttt{defResp}) \\
\texttt{defResp} & = & \lambda s.\lambda p.\text{“Not a client”} \\
\texttt{request} & = & \lambda n.\lambda s.\lambda p.f\ n\ p \\
\vdots & & \vdots
\end{array}$$

where $ccClient = \lambda n.\lambda m.((*\iota)\circ m\diamond(\lambda x.\lambda s.\lambda p.\text{“Service not available”}))\ n$

Figure 1: The Call Center object

request, $\langle\texttt{request}\rangle$, from a client is processed by selecting the value of the field `request`, which uses a function f taking as input the client number and the parameter provided from the client. As in the Abadi and Cardelli calculus, [1], methods have as first parameter a reference to `self`. (In this example this reference is not used.) The default behaviour bound to `defResp`, takes as input the parameter and returns the string “Not a client” In case the request comes from a client but it is not one of the defined requests the string “Service not available” is returned. The Call Center may

evolve to handle new requests (without changing the lookup function), or to change the policy of method selection, in which case the lookup function may change.

3 Syntax and Operational Semantics

The functional core of our calculus is a Call-By-Value lambda-calculus manipulating environments (sets of bindings between names and values). We first introduce in Section 3.1 syntax and operational semantics of the statically scoped section of the calculus which is a standard lambda-calculus with explicit substitutions. We then introduce the constructs related to freezing/defrosting expressions. In Section 3.2 we add to the calculus imperative objects.

3.1 First-Class Environments

The syntax and operational semantics for the calculus are given in Fig. 2. The expressions of the calculus, A, B, ..., in addition to basic values, **bv**, which model integers, floats *etc.*, and functions $\lambda x.A$, include bindings, that are associations between names and expressions built from the *empty environment*, (), or a *binding*, $x{=}A$ using *extension*, $A{\cdot}B$. The binding $x{=}A$ *defines* x. Extension $A{\cdot}B$ models environment evolution: the binding $x{=}B'$ in B overrides a binding for x in A. This is expressed by the congruence on environments, $\equiv$.

Free variables are defined in the standard way. (The free variables of a binding $x{=}A$ are the free variables of A.)

The *sandbox expression* $A; B$ evaluates B within the environment defined by A. Note that this implies that all the free variables of B must be defined in A, or the evaluation will lead to an error. The expression x is the lookup of x in the environment. Therefore, $(); x$ is an erroneous term since x is not bound in the environment ().

The operational semantics of this fragment of calculus is given by the relation between expressions, $A \to B$, which is defined by giving the computational steps, $\to_r$, and the reduction contexts that determine where they may happen. There are two kinds of computational steps: the first is the evaluation of an *application*, $(\lambda x.A)\ V$ which reduces to *evaluate A in the environment in which x is bound to the value V*. The second is evaluation of an expression within an environment. This *pushes the environment into*

Expressions $A, B ::= \mathbf{bv} \mid \lambda x.A \mid () \mid x{=}A \mid A{\cdot}B \mid A\ B \mid A; B \mid x$

Values $E, F ::= () \mid x{=}V \mid E{\cdot}F \qquad U, V ::= E \mid \mathbf{bv} \mid \lambda x.A$

Reduction Contexts $\mathcal{C} ::= [\,] \mid x{=}\mathcal{C} \mid \mathcal{C}{\cdot}A \mid V{\cdot}\mathcal{C} \mid \mathcal{C}\ A \mid V\ \mathcal{C} \mid \mathcal{C}; A$

Congruence of environments

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}l} ()\cdot E & \equiv & E & (x{=}U)\cdot(x{=}V) & \equiv & x{=}V & \\ E\cdot() & \equiv & E & (x{=}U)\cdot(y{=}V) & \equiv & (y{=}V)\cdot(x{=}U) & \textit{if } x \neq y \end{array}$$

$$(E\cdot E')\cdot F \equiv E\cdot(E'\cdot F)$$

Reductions: application, and nested reductions

$$\begin{array}{rcl@{\qquad}l@{\qquad}l} (\lambda x.A)\ V & \to_r & (x{=}V); A & \textit{if } \mathsf{FV}(\lambda x.A) = \emptyset & (\mathsf{app}) \\ \mathcal{C}[A] & \to & \mathcal{C}[B] & \textit{if } A \to_r B & (\mathsf{cont}) \end{array}$$

Reductions: substitution

$$\begin{array}{rcl@{\qquad}l@{\qquad}l} E; () & \to_r & () & & (\mathsf{eptS}) \\ E; \mathbf{bv} & \to_r & \mathbf{bv} & & (\mathsf{conS}) \\ E; (x{=}A) & \to_r & x{=}(E; A) & & (\mathsf{bindS}) \\ E; \lambda x.A & \to_r & \lambda x.((E{\cdot}x{=}x); A) & \textit{if } x \notin \mathsf{FV}(E) & (\mathsf{absS}) \\ E; (A{\cdot}B) & \to_r & (E; A){\cdot}(E; B) & & (\mathsf{extS}) \\ E; (A\ B) & \to_r & (E; A)\ (E; B) & & (\mathsf{callS}) \\ E; (A; B) & \to_r & (E; A); B & & (\mathsf{sbS}) \\ E; x & \to_r & V & \textit{if } E \equiv E'{\cdot}(x{=}V) & (\mathsf{varS}) \end{array}$$

Figure 2: Lambda Calculus with Environments

the expression, replacing variables by their bindings, according to the rule (varS).

Example. Let $E = (\mathsf{true}{=}(\lambda x.(\lambda y.x)){\cdot}\mathsf{false}{=}(\lambda x.(\lambda y.y)))$. This environment provides definitions for the abstractions true and false. We evaluate the expression true 3 4 in this environment, assuming integers as basic values,

as follows (we skip some trivial reduction steps to aid readability):

$$\begin{array}{rcl} E;\mathsf{true}\ 3\ 4 & \rightarrow & (E;\mathsf{true})\ (E;3)\ (E;4) \\ & \rightarrow & (\lambda x.(\lambda y.x))\ 3\ 4 \\ & \rightarrow & x{=}3;(\lambda y.x)\ 4 \\ & \rightarrow & (x{=}3;\lambda y.x)\ (x{=}3;4) \\ & \rightarrow & (\lambda y.(x{=}3{\cdot}y{=}y;x))\ 4 \\ & \rightarrow & (\lambda y.3)\ 4 \\ & \rightarrow & y{=}4;3 \\ & \rightarrow & 3 \end{array}$$

Note how an application is evaluated not by direct substitution of variables as in the classical lambda calculus, but by explicitly building an environment within which the body of the lambda is evaluated. □

The only non-obvious rules of Fig. 2 are (absS) and (sbS). In rule (absS) the variable x cannot be free in E, otherwise it would be captured by the λ-binding. (This can be always achieved by renaming the variable bound by λ.) Moreover, the environment E is extended with the binding $x{=}x$, so that A can contain free references to x. (Remember that in a sandbox expression the environment should close the expression.) The rule (sbS) for substitution in a sandbox expression, $A;B$, says that the substitution only affects the environment A, since B must be closed by A.

In Fig. 3 we introduce the additions to the syntax and operational semantics to include *frozen expressions*, $\langle A\rangle$, and their *conditional execution.* Frozen expressions are values, *e.g.*, $\langle x\rangle$ is a value whereas x is not. The reduction contexts specify that for an expression $A'{\circ}A''{\diamond}B$ we first evaluate A', and then A''. We expect that A' evaluates to an environment E and A'' to a frozen expression $\langle A\rangle$. In the reduction rules the set $\mathsf{DV}(E)$ is the set of variables defined by E, that is defined by: $\mathsf{DV}(()) = \emptyset$, $\mathsf{DV}(x{=}V) = \{x\}$, and $\mathsf{DV}(E{\cdot}E') = \mathsf{DV}(E) \cup \mathsf{DV}(E')$. If the free variables of A are all defined by E, then $E{\circ}\langle A\rangle{\diamond}B$ reduces to the sandbox expression $E;A$, rule (defOK), otherwise it reduces to B, rule (defEXC). The rule for pushing the environment in a frozen expression does not do anything since a frozen expression does not contain free variables.

Example. Consider the expression A to be $(\lambda z.(y{=}3){\circ}z{\diamond}5)\ \langle y\rangle$, where we again assume that we have integers as basic values. The evaluation of this expression is shown in Fig. 4. Note that the expression $(\lambda y.A)\ 7$ that is

$$(\lambda y.(\lambda z.(y{=}3){\circ}z{\diamond}5)\ \langle y\rangle)\ 7$$

$$
\begin{array}{rcll}
\multicolumn{4}{c}{\text{Expressions} \quad \cdots \mid \langle A\rangle \mid A{\circ}B{\diamond}B' \qquad\qquad \text{Values} \quad \cdots \mid \langle A\rangle} \\
\multicolumn{4}{c}{\text{Reduction Contexts} \quad \cdots \mid \mathcal{C}{\circ}A{\diamond}B \mid V{\circ}\mathcal{C}{\diamond}B} \\
\multicolumn{4}{c}{\text{Reductions: execution of frozen expressions}} \\
E{\circ}\langle A\rangle{\diamond}B & \rightarrow_r & E;A & \textit{if } \mathsf{FV}(A) \subseteq \mathsf{DV}(E) \quad (\mathsf{defOK}) \\
E{\circ}\langle A\rangle{\diamond}B & \rightarrow_r & B & \textit{if } \mathsf{FV}(A) \not\subseteq \mathsf{DV}(E) \quad (\mathsf{defEXC}) \\
\multicolumn{4}{c}{\text{Reductions: substitution}} \\
E;\langle A\rangle & \rightarrow_r & \langle A\rangle & (\mathsf{frS}) \\
E;(A{\circ}B{\diamond}B') & \rightarrow_r & (E;A){\circ}(E;B){\diamond}(E;B') & (\mathsf{defS})
\end{array}
$$

Figure 3: Adding Freezing/Defrosting

$$
\begin{array}{rcll}
A & \rightarrow & z{=}\langle y\rangle;(y{=}3){\circ}z{\diamond}5 & \text{by } (\mathsf{app}) \\
 & \rightarrow & (z{=}\langle y\rangle;(y{=}3)){\circ}(z{=}\langle y\rangle;z){\diamond}(z{=}\langle y\rangle;5) & \text{by } (\mathsf{defS}) \\
 & \rightarrow & (y{=}(z{=}\langle y\rangle;3)){\circ}(z{=}\langle y\rangle;z){\diamond}(z{=}\langle y\rangle;5) & \text{by } (\mathsf{bindS}) \\
 & \rightarrow & (y{=}3){\circ}(z{=}\langle y\rangle;z){\diamond}(z{=}\langle y\rangle;5) & \text{by } (\mathsf{conS}) \text{ and } (\mathsf{cont}) \\
 & \rightarrow & (y{=}3){\circ}\langle y\rangle{\diamond}(z{=}\langle y\rangle;5) & \text{by } (\mathsf{varS}) \\
 & \rightarrow & y{=}3;y & \text{by } (\mathsf{defOK}) \\
 & \rightarrow & 3 & \text{by } (\mathsf{varS})
\end{array}
$$

Figure 4: Example of Reduction

also evaluates to 3, since the y in $\langle y\rangle$ is not bound by the lambda that contains it. Variables in frozen expressions are like global variables that are dynamically bound by the environment in which they are defrosted, similar to the *special* variables of Common Lisp [25]. □

3.2 Imperative Objects

In this section we add to the calculus imperative objects. The syntax and operational semantics of the new constructs are given in Fig. 5.

Objects are *created* with the $\texttt{new}(A)$ expression that takes an environment, allocates its value in the store (heap) and returns a fresh *reference* ι to it. Given an expression A evaluating to a reference ι, the *dereferencing* expression $*A$ returns the value associated with ι in the store. Note that references ι are not part of the source language, but are needed in the expression language since they are generated during reduction. In the *object*

Expressions $\quad \cdots \mid \texttt{new}(A) \mid *A \mid A \cdot\!= B \mid A\ m(B)$

Run-time expressions $\quad \iota \mid E^\frown A\ m(B) \mid (x)^E$

Values $\quad \cdots \mid \iota$

Reduction Contexts

$\cdots \mid \texttt{new}(\mathcal{C}) \mid *\mathcal{C} \mid \mathcal{C} \cdot\!= A \mid \iota \cdot\!= \mathcal{C} \mid \mathcal{C}\ m(A) \mid E\ m(\mathcal{C}) \mid E^\frown \mathcal{C}\ m(A) \mid E^\frown \iota\ m(\mathcal{C})$

Store (maps references to environment) $\quad \sigma : \{\iota_1 \mapsto E_1, \ldots, \iota_n \mapsto E_n\}$

Reductions

$$\begin{array}{rcll}
\texttt{new}(E), \sigma & \rightarrow_r & \iota, \sigma[\iota \mapsto E] \quad \iota \textit{ is fresh} & \text{(new)} \\
*\iota, \sigma & \rightarrow_r & \sigma(\iota), \sigma & \text{(deref)} \\
\iota \cdot\!= E, \sigma & \rightarrow_r & \iota, \sigma[\iota \mapsto \sigma(\iota) \cdot E] & \text{(evolve)} \\
\iota\ m(V), \sigma & \rightarrow_r & ()^\frown \iota\ m(V), \sigma & \text{(addSr)} \\
E^\frown \iota\ m(V), \sigma & \rightarrow_r & (\lambda b.(b)^{E'}\ \iota\ V)\ (V'\ E\ \iota\ \langle m \rangle), \sigma & \text{(send)} \\
 & & \textit{where } b \textit{ is fresh and } \sigma(\iota) \equiv F \cdot (\texttt{ctx} \cdot\!= E') \cdot (\texttt{lkp} \cdot\!= V') &
\end{array}$$

Reductions: substitution

$$\begin{array}{rcll}
E; (x)^F, \sigma & \rightarrow_r & [V]^F, \sigma \quad \textit{if } E \equiv E' \cdot (x \cdot\!= V) & \text{(varRTS)} \\
E; \texttt{new}(A), \sigma & \rightarrow_r & \texttt{new}(E; A), \sigma & \text{(newS)} \\
E; \iota, \sigma & \rightarrow_r & \iota, \sigma & \text{(objS)} \\
E; *A, \sigma & \rightarrow_r & *(E; A), \sigma & \text{(derefS)} \\
E; (A \cdot\!= B), \sigma & \rightarrow_r & (E; A) \cdot\!= (E; B), \sigma & \text{(evolS)} \\
E; (A\ m(B)), \sigma & \rightarrow_r & (E; A)\ m(E; B), \sigma & \text{(sendS)} \\
E; (F^\frown A\ m(B)), \sigma & \rightarrow_r & (E; F)^\frown (E; A)\ m(E; B), \sigma & \text{(sendRTS)}
\end{array}$$

Figure 5: Adding Objects

evolution expression, $A \cdot\!= B$, the environment associated with the reference contained in the environment which is the value of A is extended with the environment which is the value of B. In a *message send*, $A\ m(B)$, the message m, with parameter the value of B, is sent to the object referenced by the value of A. In the expression $E^\frown A\ m(B)$ the environment E contains the information about the sender of the message, to be determined at run-time. As we can see from the syntax, E is not part of the source language. In fact, E is generated by the reduction rules to keep into account the context information on the sender of a message. The run-time expression $(x)^E$ stands for a variable that will be bound to a method body in which E will be added as sender to message sends.

To take into account the imperative nature of the language, the con-

figurations that are reduced are pairs of the form (*expression*, *store*), where the *store* is a mapping from references to environment values. We assume that the store is added to the configurations of the operational semantics rules of Figs. 2 and 3 and that these rules do not modify or use the store.

In rule (new) a fresh reference ι is associated in the store to the environment value E, and ι with the modified store are returned. Dereferencing returns the environment associated in the store with the reference ι, rule (deref). Object evolution, rule (evolve), extends the environment associated with the reference ι with the environment E. The reference is returned and the store is updated. Rule (addSr) adds the empty environment as sender of the method calls which are at the top level, *i.e.*, that do not appear inside the bodies of method calls.

Rule (send) specifies the reduction for message send, and it is the heart of our reduction. We assume that objects that may receive (and send) messages have two special fields: `lkp` bound to a *lookup function*, and `ctx` containing the context information for the current receiver.

The lookup function specifies how to search for the method body in response to the message m. For instance, for delegation based inheritance we first search in the current object a field m and if it is not present we continue the search in the delegate object, that is referred from a field. Similarly for class based inheritance, where an object instance of a class does not contain its methods that are instead contained in the object, representing the metaclass of the class. When creating an instance object we add a lookup function that starts the search for the field m in the metaclass of the class of the object. The lookup function does not depend on the specific object but it assumes that the object contains a field referring to the object representing the metaclass. The object metaclass will have a lookup function, which behaves similarly to the delegation based lookup, starting the search for m in the current object, and then if not found it continues the search in the object representing the metaclass of its superclass.

Regarding the context information we only specify that this field contains an environment. (This information may be used in the loookup function.) So in rule (send):

$$E\overset{\frown}{\iota}\ m(V), \sigma \rightarrow_r (\lambda b.(b)^{E'}\ \iota\ V)\ (V'\ E\ \iota\ \langle m \rangle), \sigma$$

where b is fresh and $\sigma(\iota) \equiv F{\cdot}\texttt{ctx}{=}E'{\cdot}\texttt{lkp}{=}V'$, message m is sent to the object referenced by ι which must have a field `lkp` bound to a lookup function, V', and a field `ctx` containing E', the context information for the current

receiver. Let V'' be the result of the evaluation of $(V'\ E\ \iota\ \langle m\rangle)$, that is application of the lookup function V' to the information about the context of the sender contained in E, the receiver object ι, and a frozen expression containing the name of the message to be sent; V'' is the method that must be evaluated (in response to the message). As in the Abadi-Cardelli object calculus [1], a method is a function taking as first argument the receiver object and then the parameter. We model methods with just one parameter, however since parameters may be environments this is not restrictive. The context information for the current receiver E' becomes the decoration of the variable b, and therefore will provide the sender information of all calls which occur in the method V'' (see the rule (varRTS)).

The rules for substitution are all straightforward except for (varRTS) in which $(x)^F$ in the environment $E'\cdot(x{=}V)$ is substituted by $[V]^F$, namely V where F is added as sender to the message send expressions inside V. The definition of $[V]^F$ by induction on V is given in Fig. 6. The only relevant clause is the last one, that adds F as the context information to the method call. Note that $(x)^F$ is generated at run time by rule (send), (it is not a

$$
\begin{array}{lcllcl}
[x{=}B]^F & = & x{=}[B]^F & [B{\cdot}C]^F & = & [B]^F{\cdot}[C]^F \\
[B{\cdot}{=}C]^F & = & [B]^F{\cdot}{=}[C]^F & [B;C]^F & = & [B]^F;[C]^F \\
[\lambda x.B]^F & = & \lambda x.[B]^F & [B\ C]^F & = & [B]^F\ [C]^F \\
[{*}B]^F & = & {*}[B]^F & [\langle B\rangle]^F & = & \langle [B]^F\rangle \\
[\mathtt{new}(B)]^F & = & \mathtt{new}([B]^F) & [B{\circ}C{\diamond}D]^F & = & [B]^F{\circ}[C]^F{\diamond}[D]^F \\
\multicolumn{6}{c}{[B\ m(C)]^F\ =\ E{\frown}[B]^F\ m([C]^F)}
\end{array}
$$

Figure 6: Definition of $[A]^F$

user expression) as we can see from the following example. For example $[\lambda s\lambda v.s\ m(v)]^F = \lambda s\lambda v.F{\frown}s\ m(v)$.

Example. Let $\sigma = \{\iota_1 \mapsto E_1, \iota_2 \mapsto E_2\}$ where E_1 is obtained by adding $\mathtt{ctx} = ()$ to the Call Center object of Fig. 1 and E_2 is defined by

$$
\begin{array}{rcl}
\mathtt{lkp} & = & \lambda w.\lambda s.\lambda m.({*}s){\circ}m{\diamond}(\lambda s.\lambda p.\text{“No such method”}) \\
\mathtt{ctx} & = & (\mathtt{client}{=}N) \\
\mathtt{call} & = & \lambda s.\lambda p.\ \iota_1\ \mathtt{request}(p)
\end{array}
$$

Let

- V_1 be $\lambda w.\lambda s.\lambda m.w \circ \langle ccClient\ \texttt{client}\ m \rangle \diamond (*s; \texttt{defResp})$ (the lookup function of the Call Center of Fig. 1),
- V_2 be $\lambda w.\lambda s.\lambda m.(*s) \circ m \diamond (\lambda s.\lambda p.$"No such method") (the lookup function of the client), and E be $\texttt{client}{=}N$ (the context of the client).

The relevant steps in the evaluation of $\iota_2\ \texttt{call}(V)$ with store σ, are as follows:

$$
\begin{array}{rcl}
\iota_2\ \texttt{call}(V), \sigma & \rightarrow & ()\frown\iota_2\ \texttt{call}(V), \sigma \quad \text{by (addSr)} \\
& \rightarrow & (\lambda b.(b)^E\ \iota_2\ V)\ (V_2\ ()\ \iota_2\ \langle \texttt{call} \rangle), \sigma \quad \text{by (send)} \\
& \rightarrow^* & (\lambda b.(b)^E\ \iota_2\ V)\ ((*\iota_2) \circ \langle \texttt{call} \rangle \diamond \cdots), \sigma \\
& \rightarrow^* & (\lambda b.(b)^E\ \iota_2\ V)\ (E_2; \texttt{call}), \sigma \quad \text{since } \texttt{call} \text{ is defined in } E_2 \\
& \rightarrow & (\lambda b.(b)^E\ \iota_2\ V)\ (\lambda s.\lambda p.\iota_1\ \texttt{request}(p)), \sigma \quad \text{by (varS)} \\
& \rightarrow & (b{=}\lambda s.\lambda p.\iota_1\ \texttt{request}(p)); (b)^E\ \iota_2\ V, \sigma \quad \text{by (app)} \\
(*) & \rightarrow_r^* & (\lambda s.\lambda p.\ E\frown\iota_1\ \texttt{request}(p))\ \iota_2\ V, \sigma \quad \text{by (varRTS) and Fig.6} \\
& \rightarrow^* & E\frown\iota_1\ \texttt{request}(V), \sigma \\
& \rightarrow & (\lambda b.(b)^()\ \iota_1\ V)\ (V_1\ E\ \iota_1\ \langle \texttt{request} \rangle), \sigma \quad \text{by (send)} \\
& \rightarrow^* & (\lambda b.(b)^()\ \iota_1\ V)\ (E \circ \langle ccClient\ \texttt{client}\ \langle \texttt{request} \rangle \rangle \diamond \cdots), \sigma \\
& \rightarrow & (\lambda b.(b)^()\ \iota_1\ V)\ (E; (ccClient\ \texttt{client}\ \langle \texttt{request} \rangle)), \sigma \\
& & \quad \text{since } \texttt{client} \text{ is defined in } E \\
& \rightarrow^* & (\lambda b.(b)^()\ \iota_1\ V)\ (ccClient\ N\ \langle \texttt{request} \rangle), \sigma \\
& \rightarrow^* & (\lambda b.(b)^()\ \iota_1\ V)\ ((*\iota_1) \circ \langle \texttt{request} \rangle \diamond \cdots)\ N), \sigma \\
& \rightarrow^* & (\lambda b.(b)^()\ \iota_1\ V)\ ((E_1; \texttt{request})\ N), \sigma \\
& & \quad \text{since } \texttt{request} \text{ is defined in } E_1 \\
& \rightarrow^* & (\lambda b.(b)^()\ \iota_1\ V)\ ((\lambda n.\lambda s.\lambda p.f\ n\ p)\ N), \sigma \\
& \rightarrow^* & (\lambda b.(b)^()\ \iota_1\ V)\ (\lambda s.\lambda p.f\ N\ p), \sigma \\
& \rightarrow & (b{=}\lambda s.\lambda p.f\ N\ p); (b)^()\ \iota_1\ V, \sigma \quad \text{by (app)} \\
(\star) & \rightarrow^* & (\lambda s.\lambda p.f\ N\ p)\ \iota_1\ V, \sigma \quad \text{by (varRTS) and Fig. 6} \\
& \rightarrow^* & f\ N\ V
\end{array}
$$

Reductions $(*)$ and $(\star)$ add the context information on the sender to the message send expressions in the expression bound to b. Note that in $(\star)$ there is no message send so the expression is not changed.

Assume instead that the answering policy of the Call Center is the standard delegation. That is, first see if there is a method bound to the field `request`, if not delegate the answer to an object *Delegate*, referred to by the field `delegate`. For this, the object representing the Call Center of of Fig. 1 must contain a field `delegate` whose value is the reference to its delegate object. The lookup function of the Call Center, that is the value

associated with `lkp` field would be:

$$L_d = \lambda w.\lambda s.\lambda m.(*s) \circ m \diamond (((*D); \texttt{lkp})\ w\ D\ m) \tag{3}$$

where

- D is the expression denoting a reference to the delegate of the Call Center, that is $D = *s; \texttt{delegate}$,
- w is the context information of the sender (E in the previous example),
- s is the reference to the receiver, in this case the Call Center object, and
- m is the frozen name of the message ($\langle \texttt{request} \rangle$).

If the environment $*s$, the Call Center object, contains a binding for the name contained in the frozen expression m, in this case `request`, then the associated value is returned. Otherwise, we assume that the delegate object (referred by D) has a field `lkp` containing a lookup function and $(*D); \texttt{lkp}$ evaluates to it. This lookup function is applied to w, D, and m.
Note that delegation is realized in a transparent way, since even when the method body is found in the delegate object the context information E of the sender will still appear as sender of all calls inside the body.

It is possible to combine the lookup function of Fig. 1 with delegation so that the Call Center will serve requests coming from clients as in Fig. 1, and otherwise behave as in (3).

The new lookup function is:

$$L_c = \lambda w.\lambda s.\lambda m.w \circ \langle ccClient\ \texttt{client}\ m \rangle \diamond (L_d\ w\ s\ m).$$

Similarly one can easily write lookup functions which implement class-based and trait-based searches of method bodies.

4 Type Assignment System

4.1 Types

In this section we introduce a type system for our calculus. As usual [23] (Subsection 8.1), the shapes of types are suggested by the shapes of values. We have *basic types* for basic values, *arrow types* for λ-abstractions, *reference types* for object references.

The standard typing of a binding $x = V$ is x^{ψ}, where ψ is the type of V [23] (Subsection 11.8). Since we are interested in expressing that a variable should be bound only to values of a fixed type, also in absence of a binding, we allow *binding types* of the shape $x^{\dagger\psi}$, where $\dagger \in \{!, ?\}$ is the *modality*. The meaning of $x^{!\psi}$ is that x is actually bound to a value of type ψ, while $x^{?\psi}$ says that x can only be bound to a value of type ψ. We say that x is the *subject* and ψ is the *predicate* of $x^{\dagger\psi}$. The type of an environment (*environment type*) is a set of binding types with different subjects. The empty environment is naturally typed by the empty set. Note that environment types are sets of binding types, while environments are sequences of bindings.

A frozen expression requires its set of free variables to be bound with values of fixed types: for this reason we type a frozen expression with a pair $\langle \Gamma, \psi \rangle$ (*frozen type*), whose first component Γ is a set of type assumptions for variables and whose second component ψ is the type we can derive for the expression under the assumptions in Γ.

To sum up, we introduce the five *kinds* of types, ψ, ϕ, shown in Fig. 7, where Γ is an environment type which contains only binding types with ! annotations.

For environment types, we allow *recursive types* in order to type circular object structures, and also to type the application of a method body stored in a given object to a reference to the object itself. As usual recursive types are considered modulo fold/unfold. Fig. 7, where $\dagger \in \{!, ?\}$, defines environment types, τ, ν. An *environment type* is *well formed* if all types occurring in it are well formed, it does not contain (modulo unfolding of recursive types) two binding types with the same subject. For example $x^{!\psi_1}, y^{?\psi_2}$ is well formed if ψ_1, ψ_2 are well formed, while $\mu \mathtt{t}.x^{!\mathtt{t}}, x^{?\psi}$ is not well formed. The *domain* of an environment type τ, notation $dom(\tau)$, is $\{x \mid x^{\dagger\psi} \in \tau\}$.

With $\overline{x^{\dagger\psi}}$ we abbreviate $x_1^{\dagger_1\psi_1}, \ldots, x_n^{\dagger_n\psi_n}$, $n \geq 0$. We use $\overline{x^{!\psi}}$ to indicate that the annotation of all the variables is !, similarly for ?. Let x^{ψ} be short for $x^{!\psi}$.

4.2 Typing Judgements and Rules

As usual with calculi which deal with references, the typing judgements depend on two environments: a *store environment* Σ which associates object references to types and a *standard environment* Γ which associates variables to types [23] (Section 13.4). Then we define

$$
\begin{array}{rcll}
\psi, \phi & ::= & \kappa & \textit{Basic type} \\
& | & \psi \to \psi & \textit{Arrow type} \\
& | & \mathtt{ref}\tau & \textit{Reference type} \\
& | & \langle \Gamma, \psi \rangle & \textit{Frozen type} \\
& | & \tau & \textit{Environment type} \\
\tau, \nu & ::= & x^{\dagger\psi} & \textit{Binding type} \\
& | & \tau, \tau & \textit{Sequence type} \\
& | & \mathtt{t} & \textit{Type variable} \\
& | & \mu\mathtt{t}.\tau & \textit{Recursive type}
\end{array}
$$

Figure 7: Kinds of Types and Environment Types

$$
\begin{array}{rcl}
\Sigma & = & \overline{\iota : \tau} \\
\Gamma & = & \overline{x^{!\psi}}.
\end{array}
$$

Note that a not empty standard environment is an environment type in which all modalities are !.

The typing judgement:

$$\Sigma; \Gamma \vdash A : \psi$$

says that under the environments Σ and Γ the expression A has type ψ. In the following we present and comment some significant rules. The rest of the rules can be found in Fig. 8.

We first consider the rules concerning bindings and environment extensions.

$$\frac{\Sigma; \Gamma \vdash A : \psi}{\Sigma; \Gamma \vdash x = A : x^{\dagger\psi}}\ (\mathsf{Tbind}) \qquad \frac{\Sigma; \Gamma \vdash A : \tau \quad \Sigma; \Gamma \vdash B : \tau' \quad \tau \text{ and } \tau' \text{ compatible}}{\Sigma; \Gamma \vdash A \cdot B : \tau \cdot \tau'}\ (\mathsf{Text})$$

For typing a binding we require that the expression bound to x has type ψ in order to derive the binding type $x^{\dagger\psi}$. Note that the annotation could be either ! or ?.

Two environment types τ and τ' are *compatible* if for $x \in dom(\tau) \cap dom(\tau')$ we have that x has the same predicate in τ and τ' with possibly different annotations. For example $x^{!\psi_1}, y^{?\psi_2}$ and $x^{?\psi_1}$ are compatible, while they are not compatible with $x^{!\psi_2}, y^{?\psi_2}$ if ψ_1 is not ψ_2.

$$\frac{}{\Sigma;\Gamma \vdash () : \emptyset}\ (\mathsf{Tempty}) \qquad \frac{}{\Sigma;\Gamma \vdash \mathbf{bv} : \kappa}\ (\mathsf{TBV})$$

$$\frac{\Sigma;\Gamma{\cdot}x^{\psi} \vdash A : \phi}{\Sigma;\Gamma \vdash \lambda x.A : \psi \to \phi}\ (\mathsf{Tabs}) \qquad \frac{\Sigma;\Gamma \vdash A : \phi \to \psi \quad \Sigma;\Gamma \vdash B : \phi}{\Sigma;\Gamma \vdash A\ B : \psi}\ (\mathsf{Tapp})$$

$$\frac{x^{\psi} \in \Gamma}{\Sigma;\Gamma \vdash x : \psi}\ (\mathsf{Tvar}) \qquad \frac{x^{\psi} \in \Gamma \quad \Sigma;\Gamma \vdash E : \varpi}{\Sigma;\Gamma \vdash (x)^{E} : \psi}\ (\mathsf{TvarRT})$$

$$\frac{\Sigma;\Gamma \vdash A : \tau}{\Sigma;\Gamma \vdash \mathtt{new}(A) : \mathtt{ref}\tau}\ (\mathsf{Tnew}) \qquad \frac{\iota : \tau \in \Sigma}{\Sigma;\Gamma \vdash \iota : \mathtt{ref}\tau}\ (\mathsf{Tref}) \qquad \frac{\Sigma;\Gamma \vdash A : \mathtt{ref}\tau}{\Sigma;\Gamma \vdash *A : \tau}\ (\mathsf{Tderef})$$

$$\frac{\begin{array}{c}\Sigma;\Gamma \vdash A : \mathtt{ref}\tau \quad \Sigma;\Gamma \vdash B : \psi' \quad \tau = \mu\mathtt{t}.m^{\dagger\psi}, \mathtt{lkp}^{!\phi}, \mathtt{ctx}^{!\varpi}, \tau' \\ \psi = \mathtt{reft} \to \psi' \to \psi'' \quad \phi = \varpi \to \mathtt{reft} \to \langle m^{\psi}, \psi\rangle \to \psi \quad \mathtt{t}\text{ not in }\psi'\end{array}}{\Sigma;\Gamma \vdash A\ m(B) : \psi''[\tau/\mathtt{t}]}\ (\mathsf{Tmes})$$

Figure 8: Some Typing Rules

The *extension*, $\tau{\cdot}\tau'$, of the environment types τ and τ' is defined — if τ and τ' are compatible — as the set-theoretic union of the two binding types, in case two bindings share the same subject (they must have the same predicate by definition of compatibility) we take as annotation the upper bound of the two annotations defined by: if $\dagger = \dagger' =?$, then $\dagger \sqcup \dagger' =?$ else $\dagger \sqcup \dagger' =!$. That is in the resulting environment type all the fields that were defined in one of the environment types are defined.
The environment extension is typed by the extension of the environment types.

With rule (Tsub) that follows, to an environment type τ we can add any binding with annotation ? for variables that are not already defined in τ.

$$\frac{\Sigma;\Gamma \vdash A : \tau \qquad \tau \sqsubseteq \tau'}{\Sigma;\Gamma \vdash A : \tau'}\ (\mathsf{Tsub})$$

where the subtyping relation, $\sqsubseteq$, between environment types is the reflexive and transitive closure of:

$$\frac{\phi \sqsubseteq \phi' \quad x \notin dom(\phi')}{\phi \sqsubseteq \phi', x^{?\psi}}\ (\mathsf{envAS})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\emptyset;\Gamma \vdash s : \mathtt{ref}\tau}{\emptyset;\Gamma \vdash *s : \tau} \quad \emptyset;\Gamma \vdash m : \langle m^{\psi}, \psi\rangle}{\emptyset;\Gamma \vdash (*s)\circ m \diamond A\ w\ (*s;\mathtt{d})\ m : \psi} \quad \dfrac{\mathcal{D} \quad \emptyset;\Gamma \vdash m : \langle m^{\psi}, \psi\rangle}{\emptyset;\Gamma \vdash A\ w\ (*s;\mathtt{d})\ m : \psi}}{\emptyset;\Gamma \vdash (*s)\circ m \diamond A\ w\ (*s;\mathtt{d})\ m : \psi}}{\emptyset;\{w : \varpi, s : \mathtt{ref}\tau\} \vdash \lambda m.(*s)\circ m \diamond A\ w\ (*s;\mathtt{d})\ m : \langle m^{\psi}, \psi\rangle \to \psi}}{\emptyset;\{w : \varpi\} \vdash \lambda s.\lambda m.(*s)\circ m \diamond A\ w\ (*s;\mathtt{d})\ m : \mathtt{ref}\tau \to \langle m^{\psi}, \psi\rangle \to \psi}}{\emptyset;\emptyset \vdash \lambda w.\lambda s.\lambda m.(*s)\circ m \diamond A\ w\ (*s;\mathtt{d})\ m : \varpi \to \mathtt{ref}\tau \to \langle m^{\psi}, \psi\rangle \to \psi}$$

where

$$\mathcal{D} = \dfrac{\dfrac{\mathcal{D}' \quad \emptyset;\Gamma \vdash w : \varpi}{\emptyset;\Gamma \vdash A\ w : \mathtt{ref}\tau' \to \langle m^{\psi}, \psi\rangle \to \psi} \quad \dfrac{\emptyset;\Gamma \vdash *s : \tau \quad \emptyset;\mathtt{d}^{\mathtt{ref}\tau'} \vdash \mathtt{d} : \mathtt{ref}\tau'}{\emptyset;\Gamma \vdash *s;\mathtt{d} : \mathtt{ref}\tau'}}{\emptyset;\Gamma \vdash A\ w\ (*s;\mathtt{d}) : \langle m^{\psi}, \psi\rangle \to \psi}$$

$$\mathcal{D}' = \dfrac{\dfrac{\emptyset;\Gamma \vdash *s;\mathtt{d} : \mathtt{ref}\tau'}{\emptyset;\Gamma \vdash *(*s;\mathtt{d}) : \tau'}}{\emptyset;\Gamma \vdash A : \varpi \to \mathtt{ref}\tau' \to \langle m^{\psi}, \psi\rangle \to \psi}$$

$$A = (*(*s;\mathtt{d}));\mathtt{lkp}, \quad \mathtt{d} = \mathtt{delegate}, \quad \Gamma = w : \varpi, s : \mathtt{ref}\tau, m : \langle m^{\psi}, \psi\rangle,$$
$$\tau = \mathtt{d}^{\mathtt{ref}\tau'}, \mu \quad \tau' = \mu\mathtt{t}.\mathtt{lkp}^{\phi}, \nu, \quad \phi = \varpi \to \mathtt{reft} \to \langle m^{\psi}, \psi\rangle \to \psi.$$

Figure 9: A Typing of the Lookup Function L_d

In the rule for sandbox

$$\dfrac{\Sigma;\Gamma \vdash A : \tau \quad \Sigma;\{x^{!\phi} \mid x^{!\phi} \in \tau\} \vdash B : \psi}{\Sigma;\Gamma \vdash A;B : \psi}\ (\mathsf{Tsandbox})$$

we require that A is an environment type, and that B be typable from the environment containing only the variables that in the type for A have the annotation !, which are, from rule (Tbind), (Text) and (Tsub), the variables defined in A (with rule (Tsub) we can only add variables with annotation ?). That is, the free variable of B must be defined in A.

The rules for frozen expressions and their conditional execution are as follows.

$$\frac{\emptyset;\Gamma \vdash A : \psi}{\Sigma;\Gamma' \vdash \langle A\rangle : \langle \Gamma, \psi\rangle}\ (\mathsf{Tfreeze}) \qquad \frac{\begin{array}{l}\Sigma;\Gamma \vdash A : \tau \quad \Sigma;\Gamma \vdash B : \langle \Gamma', \psi\rangle \\ \Sigma;\Gamma \vdash B' : \psi \quad dom(\tau) \supseteq dom(\Gamma') \\ \tau \text{ and } \Gamma' \text{ compatible}\end{array}}{\Sigma;\Gamma \vdash A \circ B \diamond B' : \psi}\ (\mathsf{Tdyn})$$

For a frozen expression $\langle A\rangle$, the expression A has never been reduced, and for this reason we require that no object reference occur in A. This condition is forced by the assumption that the store environment for typing A is empty. Instead the standard environment for typing A is packed with the type of A to build the frozen type of $\langle A\rangle$.

In the rule for conditional execution of a frozen expression B we require that the variables free in the frozen component which is the value of B are subjects of binding types in τ, the type of A. These variables can be typed either with annotation ! if they are defined in A or with annotation ? if they have been introduced by the rule (Tsub). Moreover, the subjects of Γ' must have the same types possibly with a different annotation in τ. This is assured by the conditions $dom(\tau) \supseteq dom(\Gamma')$ and enforcing that τ and Γ' must be compatible. We do not require (as in the rule for sandbox) that the free variable of the frozen component which is the value of B must be defined in A.

In order to type object evolution, the expression A must reduce to an object reference ι. Moreover the object stored at ι must have a type compatible with the type of B.

$$\frac{\Sigma;\Gamma \vdash A : \mathtt{ref}\tau \quad \Sigma;\Gamma \vdash B : \tau' \quad \tau \text{ and } \tau' \text{ compatible}}{\Sigma;\Gamma \vdash A := B : \mathtt{ref}(\tau \cdot \tau')}\ (\mathsf{Tevol})$$

The type of the conclusion is the type of the reference ι after the object evolution. Remember that, in our operational semantics, a binding in B overrides a field with the same name in A. For instance, we may change lookup function dynamically.

The most complex rule, as for the operational semantics, is the rule for message send:

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash A : \mathtt{ref}\tau \quad \Sigma;\Gamma \vdash B : \psi' \quad \Sigma;\Gamma \vdash E : \varpi \\ \tau = \mu \mathtt{t}.m^{\dagger\psi}, \mathtt{lkp}^{!\phi}, \mathtt{ctx}^{!\varpi}, \tau' \quad \psi = \mathtt{reft} \to \psi' \to \psi'' \\ \phi = \varpi \to \mathtt{reft} \to \langle m^{\psi}, \psi\rangle \to \psi \qquad \mathtt{t} \text{ not in } \psi'\end{array}}{\Sigma;\Gamma \vdash E \curvearrowright A\ m(B) : \psi''[\tau/\mathtt{t}]}\ (\mathsf{TmesRT})$$

In this rule we put recursive types to work. To justify the types involved in the rule we have to consider the operational semantics rule (send) and

the fact that types are preserved by reduction. Let A and B reduce to the values U and V, respectively. In order to apply rule (`send`) to $E\frown U\ m(V)$, the expression U must be a reference ι to an object, $\mathtt{ref}\tau$. This object has a field `ctx`, whose type is an environment type with only ? modalities. This environment type (denoted ϖ) holds the contextual object information that may be used by lookup functions to discriminate on the sender of a message. Moreover, the object has a field `lkp` containing the lookup function V' for the object. In order to correctly type the expression $(\lambda b.(b)^{E'}\ \iota\ V)\ (V'\ E\ \iota\ \langle m\rangle)$ obtained by reducing $E\frown\iota\ m(V)$, the lookup function V' must have a type $\phi_1 \to \phi_2 \to \phi_3 \to \phi_4$ where $\phi_1 = \varpi$ is the type of E (the sender information), ϕ_2 is the type of ι (the receiver), ϕ_3 is the type of $\langle m\rangle$ (the frozen name of the message), and ϕ_4 is the type of the method body, which is a λ-abstraction applicable first to ι (the self) and then to V (the actual parameter). Therefore, if $\mathtt{ref}\tau$ is the types of ι, then τ must be a recursive type $\mu\mathtt{t}.\cdots$ where `t` is the type of self. Then $\phi_2 = \mathtt{reft}$ since the type of the second parameter of the lookup function is the type of the receiver. Let ψ' be the type of V, the parameter of the method, and ψ'' the type of the result of the method, we have that ϕ_4, the type of the method body is $\phi_4 = \psi = \mathtt{reft} \to \psi' \to \psi''$. Note that since ψ'' may contain free occurrences of `t`, then the type in the conclusion of the rule is ψ'' where all occurrences of `t` have been replaced by τ: as usual we denote it by $\psi''[\tau/\mathtt{t}]$. Finally the type ϕ_3 of $\langle m\rangle$ is a frozen type in which in the environment m has type ψ, and the expression has type ψ. Moreover, since we want that the the lookup function may use m in a conditional expression (to search its definition) in ι we require that τ contain $m^{\dagger\psi}$ to enforce the fact that an m present in ι should be type consistent with the body found by the lookup function.

Note that we can correctly type a unique lookup function for different method types and sender types, since our type assignment system derives many types for the same untyped expressions. If we would consider a typed calculus instead we would be forced to consider polymorphic types.

Figure 9 shows a typing for the lookup function L_d as defined in (3) of Section 3.2. We assume that `t` does not occur in ϖ and ψ. For the subderivation $\mathcal{D}'$ note that $\tau' = \mathtt{lkp}^{\phi'}, \nu'$ where $\phi' = \varpi \to \mathtt{ref}\tau' \to \langle m^{\psi}, \psi\rangle \to \psi$ and ν' is the result of replacing `t` by τ' in ν.

4.3 Safety

In order to state the properties enforced by of our type system we define the *agreement* between a store environment and a store [23] [Definition 13.5.1].

Definition 1 *A store environment* Σ agrees with a memory σ *(notation* $\Sigma \vdash \sigma$*) if:*

- $\sigma(\iota) = E$ *implies* $\iota : \tau \in \Sigma$ *and* $\Sigma; \emptyset \vdash E : \tau$ *for some* τ*, and*
- $\iota : \tau \in \Sigma$ *implies* $\sigma(\iota) = E$ *and* $\Sigma; \emptyset \vdash E : \tau$ *for some* E*.*

Reducing expressions modifies the store, and for this reason also the store environment needs to evolve.

Definition 2 *We say that a store environment* Σ' *is* an evolution *of a store environment* Σ *if* $\iota : \tau \in \Sigma$ *implies* $\iota : \tau \cdot \tau' \in \Sigma'$ *for some* τ' *compatible with* τ*.*

The two results insuring that well-typed expressions do not get stuck are:

Theorem 1 (Subject Reduction) *If* $\Sigma; \Gamma \vdash A : \psi$ *and* $\Sigma \vdash \sigma$ *and* $A, \sigma \to B, \sigma'$*, then* $\Sigma'; \Gamma \vdash B : \psi$ *and* $\Sigma' \vdash \sigma'$ *for some evolution* Σ' *of* Σ*.*

Theorem 2 (Progress) *If* $\Sigma; \emptyset \vdash A : \psi$ *and* $\Sigma \vdash \sigma$ *and* A *is not a value, then there are unique* B', σ' *such that* $A, \sigma \to B, \sigma'$*.*

Subject Reduction also assures that:

- variables in an evolving environment are bound to values of fixed types;
- the free variables in the body of a sandbox are always bound in the environment of the sandbox.

5 Related Work

Abadi *et al.* were the first to study explicit substitutions as a way to bridge the gap between formal models of languages and concrete implementations [2]. The *symmetric Lisp* supports environments as first class objects, since it does not distinguish between data and programs [16]. Nishizaki developed a calculus of first-class environments in order to study dynamic software

evolution [22]. This calculus is purely functional and does not model objects or message sends.

The Piccola calculus [3] extended Milner's π-calculus [18] with first-class environments as a means to study and model software composition mechanisms. The functional core of the Piccola calculus, called the *form calculus* [20], has been used to study type inference for component-based service provision. A variant of the form calculus has also been studied by Lumpe and Schneider as a meta-framework for modeling composition mechanisms [17]. The object calculus described in the present paper can be seen as the form calculus, extended with an explicit object store, object references, message sending.

Harrison and Ossher introduced the notion of *subject-oriented programming* to acknowledge the fact that behaviour does not always depend only on the receiver of a message but also its sender [12]. Smith and Ungar demonstrated how subjectivity could be realized effectively, and how it solves numerous problems related to the context-dependent behaviour [26]. Gil and Lorenz proposed *environmental acquisition* in which objects acquire behaviour from the current containers at runtime [11]. More recently, *context-oriented programming* has emerged as a way to support *multi-dimensional dispatch* in object-oriented languages, and thus to adapt behaviour to the run-time context [14]. In the same strand [19] considers *contextual effects*, *i.e.*, the effects of the computational contexts in which expressions occur.

It is well-known that *code migration* requires dynamic reconfiguration of security policies: an interesting proposal is [13]. More difficult is modelling exchange of *open* mobile code, *i.e.*, code which may contain free variables to be bound by the receiver's code [8]. Ancona, Fagorzi and Zucca provide a combination of static and dynamic checks which assures type safety for mobile open code [4].

Type annotations are used by Damiani and Giannini [9] to discriminate whether a given field is defined or undefined in an object. Anderson and Giannini [5] used "defined/maybe" annotations on types and recursive types in an object based calculus in which fields may be added to objects. Recursive types are used, in a limited way, to type an object's "self" as well as functions returning functions. An inference algorithm has also been defined for this type system [6]. In both calculi message send is identified with method call [9] [5].

6 Concluding Remarks

We have presented a novel object calculus in which message sends are dynamically looked up, taking into account contextual information such as the identity of the sender. Objects can evolve over time, as can the lookup function itself. Method bodies may contain free variables which are dynamically bound when the method is invoked. First-class environments and "freezing" of expressions with free variables are the key mechanisms used to express dynamic binding.

Despite the highly dynamic nature of the calculus, we have demonstrated how a type assignment system can provide the usual safety guarantees.

We plan to design a type inference algorithm for the present system: this will be useful for experimenting with the present calculus without having the burden of checking typeability.

Acknowledgments

We would like to thank Mario Coppo, Ferruccio Damiani, Marcus Denker and Adrian Lienhard for their contributions to early discussions on the work presented here. Oscar Nierstrasz gratefully acknowledges the financial support of the Swiss National Science Foundation for the projects "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006-Sept. 2008) and "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010). Paola Giannini gratefully acknowledges the financial support of the MIUR PRIN'06 for the project EOS DUE.

A preliminary version of this paper was presented at the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL'08).

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[3] Franz Achermann and Oscar Nierstrasz. A calculus for reasoning about software components. *Theoretical Computer Science*, 331(2-3):367–396, 2005.

[4] Davide Ancona, Sonia Fagorzi, and Elena Zucca. A parametric calculus for mobile open code. *Electron. Notes Theor. Comput. Sci.*, 192(3):3–22, 2008.

[5] Christopher Anderson and Paola Giannini. Type checking for JavaScript. In *WOOD'05*, volume 138 of *ENTCS*, pages 37–58. Elsevier, 2005.

[6] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for Javascript. In *ECOOP'05*, volume 3586 of *LNCS*, pages 428–453. Springer, 2005.

[7] Alexandre Bergel and Stéphane Ducasse. Supporting unanticipated changes with Traits and Classboxes. In *NODe'05*, volume 69 of *LNI*, pages 61–75. GI, 2005.

[8] Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoyle, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ. In *ICFP'03*, pages 99–110. ACM, 2003.

[9] Ferruccio Damiani and Paola Giannini. Alias types for "environment-aware" computations. In *WOOD'03*, volume 82 of *ENTCS*, pages 130–150. Elsevier, 2003.

[10] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL'98*, pages 171–183. ACM Press, 1998.

[11] Joseph Gil and David H. Lorenz. Environmental acquisition - a new inheritance-like abstraction mechanism. In *OOPSLA'96*, volume 31 of *ACM SIGPLAN Notices*, pages 214–231, 1996.

[12] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *OOPSLA'93*, volume 28 of *ACM SIGPLAN Notices*, pages 411–428, 1993.

[13] Brant Hashii, Scott Malabarba, Raju Pandey, and Matt Bishop. Supporting reconfigurable security policies for mobile programs. *Computer Networks*, 33:77–93, 200.

[14] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[15] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, May 2001.

[16] Suresh Jagannathan. *A Programming Language Supporting First-Class Parallel Environments*. PhD thesis, M.I.T., 1989.

[17] Markus Lumpe and Jean-Guy Schneider. A form-based metamodel for software composition. *Journal of Science of Computer Programming*, 56(2):59–78, 2005.

[18] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, 1992.

[19] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *POPL'08*, pages 37–50. ACM, 2008.

[20] Oscar Nierstrasz. Contractual types. Technical Report IAM-03-004, Institute of Computer Science, University of Bern, Switzerland, 2003.

[21] Oscar Nierstrasz, Alexandre Bergel, Marcus Denker, Stéphane Ducasse, Markus Gaelli, and Roel Wuyts. On the revival of dynamic languages. In *Software Composition'05*, volume 3628 of *LNCS*, pages 1–13. Springer, 2005. Invited paper.

[22] Shin-ya Nishizaki. Programmable environment calculus as theory of dynamic software evolution. In *ISPSE'00*, pages 221–225. IEEE Computer Society Press, 2000.

[23] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[24] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Journal of Computer Languages, Systems and Structures*, 34(2-3):46–65, 2008.

[25] Peter Seibel. *Practical CommonLisp*. Apress, 2005.

[26] Randall B. Smith and Dave Ungar. A simple and unifying approach to subjective objects. *TAPOS special issue on Subjectivity in Object-Oriented Systems*, 2(3):161–178, 1996.

[27] David Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, February 1995.

A Appendix

A.1 Proof of Soundness

The *restriction* of an environment Γ with respect to a set of variables X, notation $\Gamma{\restriction}X$, is defined as follows

$$\Gamma{\restriction}X = \begin{cases} \emptyset & \textit{if } \Gamma = \emptyset, \\ x^{\dagger\psi}, \Gamma'{\restriction}X & \textit{if } x \in X \textit{ and } \Gamma = x^{\dagger\psi}, \Gamma' \\ \Gamma'{\restriction}X & \textit{if } x \notin X \textit{ and } \Gamma = x^{\dagger\psi}, \Gamma'. \end{cases}$$

The *restriction* of an environment Σ with respect to a set of object identifiers O, notation $\Sigma{\restriction}O$, is defined similarly.
By $\mathsf{OID}(A)$ we denote the set of object identifiers which occur in A and by $\mathsf{FV}(A)$ the set of term variables which occur free in A.

Given an environment type τ, we denote by $(\tau)^!$ the maximal environment type contained in τ in which all binding types have the ! modality, i.e. we define:

$$(\tau)^! = \begin{cases} x^{!\psi}{\cdot}(\tau')^! & \text{if } \tau = x^{!\psi}{\cdot}\tau', \\ (\tau')^! & \text{if } \tau = x^{?\psi}{\cdot}\tau'. \end{cases}$$

The proofs of the following propositions by structural induction on expressions is straightforward.

Proposition 1 *If* $\Sigma;\Gamma \vdash A : \psi$*, then* $dom(\Sigma) \supseteq \mathsf{OID}(A)$ *and* $dom(\Gamma) \supseteq \mathsf{FV}(A)$ *and* $\Sigma{\restriction}\mathsf{OID}(A);\Gamma \vdash A : \psi$ *and* $\Sigma;\Gamma{\restriction}\mathsf{FV}(A) \vdash A : \psi$.

Proposition 2 *If* A *is a closed expression then either* A *is a value, or there is a unique context* $\mathcal{C}$ *such that* $A = \mathcal{C}[\mathcal{R}]$ *for some redex* $\mathcal{R}$.

Due to the previous proposition given an expression A there is at most one rule applicable to A, so the reduction is deterministic.

By looking at the typing rules we can easily prove the following standard lemmas.

Lemma 1 (Canonical Forms) *1. If $\Sigma;\Gamma \vdash U : \emptyset$, then $U = ()$.*

2. If $\Sigma;\Gamma \vdash U : \kappa$, then $U \equiv \mathbf{bv}$ for some basic value $\mathbf{bv}$.

3. If $\Sigma;\Gamma \vdash U : x^{!\psi}, \tau$, then $U \equiv E{\cdot}(x = V)$ and $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;\Gamma \vdash V : \psi$ for some V.

4. If $\Sigma;\Gamma \vdash U : x^{?\psi}, \tau$, then either $U \equiv E{\cdot}(x = V)$ and $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;\Gamma \vdash V : \psi$ for some E, V, or $x \notin \mathsf{DV}(U)$ and $\Sigma;\Gamma \vdash U : \tau$.

5. If $\Sigma;\Gamma \vdash U : \phi \to \psi$, then $U = \lambda x.A$ and $\Sigma;\Gamma{\cdot}x^{\phi} \vdash A : \psi$ for some $\lambda x.A$.

6. If $\Sigma;\Gamma \vdash U : \mathtt{ref}\tau$, then $U = \iota$ and $\iota : \tau' \in \Sigma$ with $\tau' \sqsubseteq \tau$ for some ι, τ'.

7. If $\Sigma;\Gamma \vdash U : \langle \Gamma', \psi \rangle$, then $U = \langle A \rangle$ and $\Sigma;\Gamma' \vdash A : \psi$ for some A.

Lemma 2 (Inversion) *Let $\Sigma;\Gamma \vdash A : \psi$.*

1. If A is $()$, then $\psi = \overline{x^{?\psi}}$.

2. If A is a basic value, then $\psi = \kappa$, for some basic type κ.

3. If A is x, then for some ψ' we have that $x^{\psi'} \in \Gamma$ and $\psi' \sqsubseteq \psi$.

4. If A is $(x)^E$, then $\Sigma;\Gamma \vdash E : \varpi$ and $x^{\psi'} \in \Gamma$, for some environment type ϖ which only contains ? modalities and some ψ' such that $\psi' \sqsubseteq \psi$.

5. If A is $\langle B \rangle$, then $\psi = \langle \Gamma', \psi' \rangle$ and $\Sigma;\Gamma' \vdash B : \psi'$ for some Γ', ψ'.

6. If A is $x{=}B$, then $\psi = x^{\dagger\psi'}, \overline{x^{?\psi}}$, and $\Sigma;\Gamma \vdash B : \psi'$ for some $\psi', \overline{x^{?\psi}}$.

7. If A is $\lambda x.B$, then $\psi = \psi' \to \phi$ and $\Sigma;\Gamma{\cdot}x^{\psi'} \vdash B : \phi$ for some ψ', ϕ.

8. If A is $B{\cdot}C$, then $\psi = \tau{\cdot}\tau'$ and $\Sigma;\Gamma \vdash B : \tau$ and $\Sigma;\Gamma \vdash C : \tau'$ for some compatible τ, τ'.

9. If A is $B;C$, then $\Sigma;\Gamma \vdash B : \tau$ and $\Sigma;(\tau)^! \vdash C : \psi$ for some τ.

10. *If* A *is* $B \circ C \diamond C'$, *then* $\Sigma;\Gamma \vdash B : \tau$, *and* $\Sigma;\Gamma \vdash C : \langle \Gamma', \psi \rangle$, *and* $\Sigma;\Gamma \vdash C' : \psi$, *and* $dom(\tau) \supseteq dom(\Gamma')$, *for some compatible* τ, Γ' .

11. *If* A *is* $B\ C$, *then* $\Sigma;\Gamma \vdash B : \psi' \to \phi$ *and* $\Sigma;\Gamma \vdash C : \psi'$ *and* $\phi \sqsubseteq \psi$ *for some* ϕ, ψ'.

12. *If* A *is* ι, *then* $\psi = \mathtt{ref}\tau$ *and* $\iota : \tau' \in \Sigma$ *for some* $\tau' \sqsubseteq \tau$.

13. *If* A *is* $*B$, *then* $\Sigma;\Gamma \vdash B : \mathtt{ref}\psi$.

14. *If* A *is* $\mathtt{new}(B)$, *then* $\psi = \mathtt{ref}\tau$ *and* $\Sigma;\Gamma \vdash B : \tau$ *for some* τ.

15. *If* A *is* $B \cdot{=} C$, *then* $\psi = \mathtt{ref}(\tau \cdot \tau')$ *and* $\Sigma;\Gamma \vdash B : \mathtt{ref}\tau$ *and* $\Sigma;\Gamma \vdash C : \tau'$ *for some compatible* τ, τ'.

16. *If* A *is* $B\ m(C)$, *then* $\psi = \psi''[\tau/\mathtt{t}]$ *and* $\Sigma;\Gamma \vdash B : \mathtt{ref}\tau$ *and* $\Sigma;\Gamma \vdash C : \psi'$ *and* $\tau = \mu\mathtt{t}.m^{\dagger\phi'}, \mathtt{lkp}^{!\phi}, \mathtt{ctx}^{!\varpi}, \tau'$ *and* $\phi' = \mathtt{reft} \to \psi' \to \psi''$ *and* $\phi = \varpi \to \mathtt{reft} \to \langle m^{\phi'}, \phi' \rangle \to \phi'$ *for some* $\phi, \phi', \psi', \psi'', \varpi$ *such that* ϖ *is an environment type which only contains* ? *modalities and* $\mathtt{t}$ *does not occur in* ϕ'.

17. *If* A *is* $E \frown B\ m(C)$, *then* $\psi = \psi''[\tau/\mathtt{t}]$ *and* $\Sigma;\Gamma \vdash B : \mathtt{ref}\tau$ *and* $\Sigma;\Gamma \vdash C : \psi'$ *and* $\Sigma;\Gamma \vdash E : \varpi$ *and* $\tau = \mu\mathtt{t}.m^{\dagger\phi'}, \mathtt{lkp}^{!\phi}, \mathtt{ctx}^{!\varpi}, \tau'$ *and* $\phi' = \mathtt{reft} \to \psi' \to \psi''$ *and* $\phi = \varpi \to \mathtt{reft} \to \langle m^{\phi'}, \phi' \rangle \to \phi'$ *for some* $\phi, \phi', \psi', \psi'', \varpi$ *such that* ϖ *is an environment type which only contains* ? *modalities and* $\mathtt{t}$ *does not occur in* ψ'.

Lemma 3 (Weakening) *If* $\Sigma;\Gamma \vdash A : \psi$, *and* $\Gamma' \supseteq \Gamma$, *then* $\Sigma;\Gamma' \vdash A : \psi$.

Lemma 4 1. *If* $\Sigma;\Gamma \vdash A : \psi$, *and* $A = \mathcal{C}[\mathcal{R}]$, *then* $\Sigma;\Gamma \vdash \mathcal{R} : \psi'$ *for some* ψ'.

2. *If* $\Sigma;\Gamma \vdash \mathcal{C}[\mathcal{R}] : \psi$ *where* $\Sigma;\Gamma \vdash \mathcal{R} : \psi'$, *and* $\Sigma;\Gamma \vdash A : \psi'$, *then* $\Sigma;\Gamma \vdash \mathcal{C}[A] : \psi$.

Proof: By induction on evaluation contexts. □

Given an environment type τ, and a set of variables X, we denote by $\tau \backslash X$ the environment type obtained from τ by removing the types for the variables in X.

If $\tau \restriction X = \overline{x^{\dagger\psi}}$ we define $\tau^{(!,X)} = \overline{x^{!\psi}}$.

Lemma 5 1. *If* $\Sigma;\Gamma \vdash E : \tau$, *and* $x \in \mathsf{DV}(E)$, *then* $x^{\dagger\psi} \in \tau$ *and* $\Sigma;\Gamma \vdash E : (\tau \backslash \{x\}) \cdot x^{!\psi}$.

2. *If* $\Sigma;\Gamma \vdash E : \tau$, *and* $x \notin \mathsf{DV}(E)$, *then* $x^{\dagger\psi} \in \tau$ *and* $\Sigma;\Gamma \vdash E : \tau\backslash\{x\}$.

3. *If* $\Sigma;\Gamma \vdash E : \tau$, *then* $\Sigma;\Gamma \vdash E : \tau^{(!,\mathsf{DV}(E))}$.

Proof: By induction on E using Lemma 1(3) and (4). □

Lemma 6 *If* $\Sigma;\Gamma \vdash V : \psi$, *and* $\Sigma;\Gamma \vdash F : \varpi$, *then* $\Sigma;\Gamma \vdash [V]^F : \psi$.

Proof: By induction on V by noting that the only difference between the typing rules (Tmes) and (TmesRT) is the addition of the typing of the context F in the premise of the rule. □

Proof of Subject Reduction

If $\Sigma;\Gamma \vdash A : \psi$ *and* $\Sigma \vdash \sigma$ *and* $A, \sigma \to B, \sigma'$, *then* $\Sigma';\Gamma \vdash B : \psi$ *and* $\Sigma' \vdash \sigma'$ *for some evolution* Σ' *of* Σ.
Proof: Let first consider $A, \sigma \to_r B, \sigma'$. The proof is by cases on the operational semantics rule used. We do not mention the store when it is unmodified.

- Rule (app). In this case A is $U\ V$ and B is $x{=}V; A'$ where $U = \lambda x.A'$ and $\lambda x.A'$ is closed. Since $\Sigma;\Gamma \vdash A : \psi$ by Lemma 2(11) we have that

$$\Sigma;\Gamma \vdash U : \phi \to \phi' \tag{4}$$

$$\Sigma;\Gamma \vdash V : \phi \tag{5}$$

for some ϕ' such that $\phi' \sqsubseteq \psi$. From (4) we get $\Sigma;\emptyset \vdash \lambda x.A' : \phi \to \phi'$ by Proposition 1. Therefore from Lemma 2(7) we derive that

$$\Sigma; x^{\phi} \vdash A' : \phi'. \tag{6}$$

Applying rule (Tbind) to (5) we obtain:

$$\Sigma;\Gamma \vdash x{=}V : x^{\phi}. \tag{7}$$

Therefore from (7), (6), and rule (Tsandbox) we have that

$$\Sigma;\Gamma \vdash x{=}V; A' : \phi'.$$

Applying rule (Tsub) we derive $\Sigma;\Gamma \vdash x{=}V; A' : \psi$.

- Rule (new). In this case A is $\mathtt{new}(E)$ and B is ι and σ' is $\sigma[\iota \mapsto E]$, where ι is fresh. From Lemma 2(14) we get $\psi = \mathtt{ref}\tau$ and $\Sigma;\Gamma \vdash E : \tau$ for some τ. We can take $\Sigma' = \Sigma, \iota : \tau$ and conclude using rule (Tref) and the definition of agreement between store environments and memory.

- Rule (deref). In this case A is $*\iota$ and B is $\sigma(\iota)$. From Lemma 2(13) we get $\Sigma;\Gamma \vdash \iota : \mathtt{ref}\psi$. By Lemma 2(12) $\iota : \tau \in \Sigma$ with $\tau \sqsubseteq \psi$, which implies $\Sigma;\Gamma \vdash \sigma(\iota) : \psi$ by definition of agreement between store environments and memory, possibly using rule (Tsub).

- Rule (evolve). In this case A is $\iota \cdot = E$ and B is ι and $\sigma' = \sigma[\iota \mapsto \sigma(\iota)\cdot E]$. From Lemma 2(15) we get $\psi = \mathtt{ref}(\tau\cdot\tau')$ and $\Sigma;\Gamma \vdash \iota : \mathtt{ref}\tau$ and $\Sigma;\Gamma \vdash E : \tau'$ for some compatible τ, τ'. We take

$$\Sigma'(\iota') = \begin{cases} \tau\cdot\tau' & \text{if } \iota' = \iota, \\ \Sigma'(\iota) & \text{otherwise.} \end{cases}$$

  By rule (Tref) we get $\Sigma';\Gamma \vdash \iota : \mathtt{ref}(\tau\cdot\tau')$. Clearly Σ' is an evolution of Σ and $\Sigma \vdash \sigma$ implies $\Sigma' \vdash \sigma'$.

- Rule (addSr). This case follows easily from Lemma 2(17) and rule (TmesRT).

- Rule (send). In this case A is $E\curvearrowright\iota\ m(V)$, $\sigma(\iota) \equiv F\cdot(\mathtt{ctx}{=}E')\cdot(\mathtt{lkp}{=}V')$ and B is $(\lambda b.(b)^{E'}\ \iota\ V)\ (V'\ E\ \iota\ \langle m\rangle)$. From Lemma 2(17) we get $\psi = \psi''[\tau/\mathtt{t}]$ and $\Sigma;\Gamma \vdash \iota : \mathtt{ref}\tau$ and $\Sigma;\Gamma \vdash V : \psi'$ and $\Sigma;\Gamma \vdash E : \varpi$ and $\tau = \mu\mathtt{t}.m^{\dagger\phi'}, \mathtt{lkp}^{!\phi}, \mathtt{ctx}^{!\varpi}, \tau'$ and $\phi' = \mathtt{reft} \to \psi' \to \psi''$ and $\phi = \varpi \to \mathtt{reft} \to \langle m^{\phi'}, \phi'\rangle \to \phi'$ for some $\phi, \phi', \psi', \psi'', \varpi$ such that ϖ is an environment type which only contains ? modalities and $\mathtt{t}$ does not occur in ψ'. By Lemma 1(6) and the agreement between Σ and σ we get $\Sigma;\Gamma \vdash V' : \phi$. This implies $\Sigma;\Gamma \vdash V'\ E\ \iota\ \langle m\rangle : \phi'[\tau/\mathtt{t}]$ by rules (Tfreeze) and (Tapp). We can also derive $\Sigma;\Gamma \vdash \lambda b.(b)^{E'}\ \iota\ V : \phi'[\tau/\mathtt{t}] \to \psi''[\tau/\mathtt{t}]$, and so we conclude $\Sigma;\Gamma \vdash B : \psi''[\tau/\mathtt{t}]$ using rule (Tapp).

- Rule (defOK). In this case A is $E\circ\langle A'\rangle\diamond B'$ and B is $E; A'$ and $\mathsf{FV}(A') \subseteq \mathsf{DV}(E)$. Since $\Sigma;\Gamma \vdash A : \psi$ we have by Lemmas 2(10) and 1(7) that $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;\Gamma \vdash \langle A'\rangle : \langle\Gamma', \psi\rangle$ and $dom(\tau) \supseteq dom(\Gamma')$ for some compatible τ, Γ'. We get $dom(\tau \restriction \mathsf{DV}(E)) \supseteq dom(\Gamma' \restriction \mathsf{FV}(A'))$ which

implies $\tau^{(!,\mathsf{DV}(E))} \supseteq \Gamma' \restriction \mathsf{FV}(A')$. From Lemma 5(3) we derive $\Sigma;\Gamma \vdash E : \tau^{(!,\mathsf{DV}(E))}$. From Lemma 2(5) we derive $\Sigma;\Gamma' \vdash A' : \psi$ and by Proposition 1 $\Sigma;\Gamma' \restriction \mathsf{FV}(A') \vdash A' : \psi$, which implies $\Sigma;\tau^{(!,\mathsf{DV}(E))} \vdash A' : \psi$ by Lemma 3. We conclude $\Sigma;\Gamma \vdash E;A' : \psi$ using rule (Tsandbox).

- Rule (defEXC). In this case A is $E \circ \langle A' \rangle \diamond B'$ and B is B'. This case is easy by Lemma 2(10).

- Rule (eptS). Let A be $E;()$ and $B = ()$. Since $\Sigma;\Gamma \vdash A : \psi$ we have by Lemma 2(9) that $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;(\tau)^! \vdash () : \psi$ for some τ. By Lemma 2(1) $\Sigma;(\tau)^! \vdash () : \psi$ implies $\psi = \overline{x^{?\psi}}$. Applying rules (Tempty) and (Tsub) we get $\Sigma;\Gamma \vdash () : \overline{x^{?\psi}}$.

- Rule (conS). In this case A is $E;\mathbf{bv}$ and B is $\mathbf{bv}$. Since $\Sigma;\Gamma \vdash A : \psi$ we have by Lemma 2(9) that $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;(\tau)^! \vdash \mathbf{bv} : \psi$ for some τ. From Lemma 2(2) we get $\psi = \kappa$, so we conclude applying rule (TBV).

- Rule (bindS). In this case A is $E;x{=}A'$ and B is $x{=}(E;A')$. Since $\Sigma;\Gamma \vdash A : \psi$ we have by Lemma 2(9) $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;(\tau)^! \vdash x{=}A' : \psi$ for some τ. By Lemma 2(6) we have $\psi = \overline{x^{?\psi}}, x^{\dagger\psi'}$ and $\Sigma;(\tau)^! \vdash A' : \psi'$, and so from rule (Tsandbox) we get $\Sigma;\Gamma \vdash E;A' : \psi'$ for some $\overline{x^{?\psi}}, \psi'$. Applying rules (Tbind), (Tsub) we conclude

$$\Sigma;\Gamma \vdash x{=}(E;A') : \psi.$$

- Rule (absS). In this case A is $E;\lambda x.A'$ and B is $\lambda x.(E{\cdot}x{=}x);A'$ where $x \notin \mathsf{FV}(E)$. Since $\Sigma;\Gamma \vdash A : \psi$ we have by Lemma 2(9) that $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;(\tau)^! \vdash \lambda x.A' : \psi$ for some τ. From $\Sigma;(\tau)^! \vdash \lambda x.A' : \psi$ and Lemma 2(7) we have that $\psi = \psi' \to \psi''$ for some ψ'' and

$$\Sigma;(\tau)^!{\cdot}x^{\psi'} \vdash A' : \psi''. \tag{8}$$

From Proposition 1 and $x \notin \mathsf{FV}(E)$ we have that $\Sigma;\Gamma \restriction \{x\} \vdash E : \tau$, and from Lemma 3 $\Sigma;(\Gamma \restriction \{x\}){\cdot}x^{\psi'} \vdash E : \tau$. From $\Sigma;x^{\psi'} \vdash x{=}x : x^{\psi'}$, Lemma 3, and $(\Gamma \restriction \{x\}){\cdot}x^{\psi'} \supseteq x^{\psi'}$, applying rule (Text) we derive

$$\Sigma;\Gamma{\cdot}x^{\psi'} \vdash E{\cdot}x{=}x : \tau{\cdot}x^{\psi'}. \tag{9}$$

From (9) and (8) by (Tsandbox) we get $\Sigma;(\Gamma \restriction \{x\}){\cdot}x^{\psi'} \vdash (E{\cdot}x{=}x);A' : \psi''$. Finally by (Tabs) and Lemma 3 we conclude $\Sigma;\Gamma \vdash \lambda x.(E{\cdot}x{=}x);A' : \psi$.

- Rule (frS). In this case A is $E;\langle A'\rangle$ and B is $\langle A'\rangle$. Since $\Sigma;\Gamma \vdash A : \psi$ we have by Lemma 2(9) that $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;(\tau)^! \vdash \langle A'\rangle : \psi$ for some τ. From Lemma 2(5) we derive $\psi = \langle \Gamma', \psi'\rangle$ and $\Sigma;\Gamma' \vdash A' : \psi'$ for some ψ', Γ'. We conclude $\Sigma;\Gamma \vdash \langle A'\rangle : \psi$ using rule (Tfreeze).

- Rule (extS). In this case A is $E; A_1 \cdot A_2$ and B is $(E;A_1)\cdot(E;A_2)$. Since $\Sigma;\Gamma \vdash A : \psi$ we have by Lemma 2(9) that $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;(\tau)^! \vdash A_1 \cdot A_2 : \psi$ for some τ. From Lemma 2(8) we derive that $\Sigma;(\tau)^! \vdash A_i : \tau_i$, $i = 1, 2$ and $\psi = \tau_1 \cdot \tau_2$ for some τ_1, τ_2. Applying rule (Tsandbox) twice we have $\Sigma;\Gamma \vdash E; A_i : \tau_i$ for $i = 1, 2$. Therefore, from rule (Text) we get

$$\Sigma;\Gamma \vdash (E;A_1)\cdot(E;A_2) : \psi.$$

- Rule (callS). In this case A is $E;(A'\ B')$ and B is $(E;A')\ (E;B')$. Lemma 2(9) implies $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;(\tau)^! \vdash A'\ B' : \psi$ for some τ. By Lemma 2(11) we get $\Sigma;(\tau)^! \vdash A' : \psi' \to \psi''$ and $\Sigma;(\tau)^! \vdash B' : \psi'$ and $\psi'' \sqsubseteq \psi$ for some ψ', ψ''. By rule (Tsandbox) $\Sigma;\Gamma \vdash E; A' : \psi' \to \psi''$ and $\Sigma;\Gamma \vdash E; B' : \psi'$. We conclude using rules (Tapp) and (Tsub).

- Rule (sbS). In this case A is $E;(A';B')$ and B is $(E;A');B'$. Since $\Sigma;\Gamma \vdash A : \psi$ we have by Lemma 2(9) that $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;(\tau)^! \vdash A';B' : \psi$ for some τ. Again by Lemma 2(9) we get $\Sigma;(\tau)^! \vdash A' : \tau'$ and $\Sigma;(\tau')^! \vdash B' : \psi$ for some τ'. Applying rule (Tsandbox) we derive first $\Sigma;\Gamma \vdash E; A' : \tau'$ and then $\Sigma;\Gamma \vdash (E;A');B' : \psi$.

- Rule (defS). Here A is $E; A' \circ B_1 \diamond B_2$ and B is $(E;A')\circ(E;B_1)\diamond(E;B_2)$. Since $\Sigma;\Gamma \vdash A : \psi$ we have by Lemma 2(9) that $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;(\tau)^! \vdash A' \circ B_1 \diamond B_2 : \psi$ for some τ. From Lemma 2(10) we derive that $\Sigma;(\tau)^! \vdash A' : \tau'$ and $\Sigma;(\tau)^! \vdash B_1 : \langle \Gamma', \psi\rangle$ and $\Sigma;(\tau)^! \vdash B_2 : \psi$ and $dom(\tau') \supseteq dom(\Gamma')$ for some compatible τ, Γ'. Applying rule (Tsandbox) we have $\Sigma;\Gamma \vdash E; A' : \tau'$ and $\Sigma;\Gamma \vdash E; B_1 : \langle \Gamma', \psi\rangle$ and $\Sigma;\Gamma \vdash E; B_2 : \psi$. From rule (Tdyn) we get

$$\Sigma;\Gamma \vdash (E;A')\circ(E;B_1)\diamond(E;B_2) : \psi.$$

- Rule (varS). In this case A is $E; x$ and $E \equiv E' \cdot (x = V)$ and B is V. Since $\Sigma;\Gamma \vdash A : \psi$ we have by Lemma 2(9) that $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;(\tau)^! \vdash x : \psi$ for some τ. By Lemma 2(8) $\tau = \tau' \cdot \nu$ and $\Sigma;\Gamma \vdash E' : \tau'$ and $\Sigma;\Gamma \vdash x = V : \nu$ for some τ', ν. By Lemma 2(6) we get $\nu = x^{\psi'}$

and $\Sigma;\Gamma \vdash V : \psi'$. From $\tau = \tau' \cdot x^{\psi'}$ and $\Sigma;(\tau)^! \vdash x : \psi$ we conclude $\psi' = \psi$ by Lemma 2(3).

- Rule (varRTS). In this case A is $E;(x)^F$ and $E \equiv E' \cdot (x = V)$ and B is $[V]^F$. Since $\Sigma;\Gamma \vdash A : \psi$ we have by Lemma 2(9) that $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;(\tau)^! \vdash (x)^F : \psi$ for some τ. By Lemma 2(8) $\tau = \tau' \cdot \nu$ and $\Sigma;\Gamma \vdash E' : \tau'$ and $\Sigma;\Gamma \vdash x = V : \nu$ for some τ', ν. By Lemma 2(6) we get $\nu = x^{\psi'}$ and $\Sigma;\Gamma \vdash V : \psi'$. From $\tau = \tau' \cdot x^{\psi'}$ and $\Sigma;(\tau)^! \vdash x : \psi$, by Lemma 2(4) we have that $\psi' = \psi$, and $\Sigma;\Gamma \vdash F : \varpi$. Therefore, $\Sigma;\Gamma \vdash V : \psi$, and from Lemma 6 we derive that $\Sigma;\Gamma \vdash [V]^F : \psi$.

- Rule (newS). Here A is $E;\mathtt{new}(A')$ and B is $\mathtt{new}(E;A')$. Lemma 2(9) implies $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;(\tau)^! \vdash \mathtt{new}(A') : \psi$ for some τ. By Lemma 2(14) we get $\psi = \mathtt{ref}\tau'$ and $\Sigma;(\tau)^! \vdash A' : \tau'$ for some τ'. By rule (Tsandbox) $\Sigma;\Gamma \vdash E;A' : \tau'$, so we conclude using rule (Tnew).

- Rule (objS). In this case A is $E;\iota$ and B is ι. Lemma 2(9) implies $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;(\tau)^! \vdash \iota : \psi$ for some τ. By Proposition 1 $\Sigma;\emptyset \vdash E;\iota : \psi$ and then $\Sigma;\Gamma \vdash E;\iota : \psi$ by Lemma 3.

- Rule (derefS). In this case A is $E;*A'$ and B is $*(E;A')$. Lemma 2(9) implies $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;(\tau)^! \vdash *A' : \psi$ for some τ. By Lemma 2(13) we get $\Sigma;(\tau)^! \vdash A' : \mathtt{ref}\psi$. By rule (Tsandbox) $\Sigma;\Gamma \vdash E;A' : \mathtt{ref}\psi$, so we conclude using rule (Tderef).

- Rule (evolS). In this case A is $E;(A' \cdot\!= B')$ and B is $(E;A') \cdot\!= (E;B')$. Lemma 2(9) implies $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;(\tau)^! \vdash A'.\!=B' : \psi$ for some τ. By Lemma 2(15) we get $\psi = \mathtt{ref}(\tau' \cdot \tau'')$ and $\Sigma;(\tau)^! \vdash A' : \mathtt{ref}\tau'$ and $\Sigma;(\tau)^! \vdash B' : \tau''$ for some compatible τ', τ''. By rule (Tsandbox) we get $\Sigma;\Gamma \vdash E;A' : \mathtt{ref}\tau'$ and $\Sigma;\Gamma \vdash E;B' : \tau''$, so we conclude using rule (Tevol).

- Rule (sendS). This case is similar and simpler than the following case.

- Rule (sendRTS). In this case A is $E;(F \frown A'\ m(B'))$ and B is $(E;F) \frown (E;A')\ m(E;B')$. Lemma 2(9) implies $\Sigma;\Gamma \vdash E : \tau$ and $\Sigma;(\tau)^! \vdash F \frown A'\ m(B') : \psi$ for some τ. By Lemma 2(17) we get $\psi = \psi''[\nu/\mathtt{t}]$ and $\Sigma;(\tau)^! \vdash A' : \mathtt{ref}\nu$ and $\Sigma;(\tau)^! \vdash B' : \psi'$ and $\Sigma;(\tau)^! \vdash F : \varpi$ and $\nu = \mu\mathtt{t}.m^{\dagger\phi'}, \mathtt{lkp}^{!\phi}, \mathtt{ctx}^{!\varpi}, \nu'$ and $\phi' = \mathtt{reft} \to \psi' \to \psi''$ and $\phi = \varpi \to \mathtt{reft} \to \langle m^{\phi'}, \phi' \rangle \to \phi'$ for some $\phi, \phi', \psi', \psi'', \varpi$ such

that ϖ is an environment type which only contains ? modalities and $\mathtt{t}$ does not occur in ψ'. By rule (Tsandbox) we get $\Sigma;\Gamma \vdash E; A' : \mathtt{ref}\nu$ and $\Sigma;\Gamma \vdash E; B' : \psi'$ and $\Sigma;\Gamma \vdash E; F : \varpi$, so we conclude using rule (TmesRT).

If $A, \sigma \to B, \sigma'$, then the only applicable rule is (cont). Therefore A is $\mathcal{C}[\mathcal{R}]$, $\mathcal{R} \to_r A'$, and B is $\mathcal{C}[A']$. By Lemma 4(1) we have that $\Sigma;\Gamma \vdash \mathcal{R} : \psi'$. By the previous proof we have that $\Sigma;\Gamma \vdash A' : \psi'$, and by Lemma 4(2) we conclude that $\Sigma;\Gamma \vdash \mathcal{C}[A'] : \psi$. □

Proof of Progress

If $\Sigma;\emptyset \vdash A : \psi$ and $\Sigma \vdash \sigma$ and A is not a value, then there are unique B', σ' such that $A, \sigma \to B, \sigma'$.

Proof: From Proposition 2 and $\Sigma;\emptyset \vdash A : \psi$, we have that there is a unique $\mathcal{C}$ such that A is $\mathcal{C}[\mathcal{R}]$ for some redex $\mathcal{R}$.

Case: $\mathcal{C} = [\,]$. The proof is by cases on redexes. For most of them we can reduce applying the corresponding rule, so we only consider the cases in which the rule has some side condition.

- Case $U\ V$. By Lemma 2(11) $\Sigma;\emptyset \vdash U\ V : \psi$ implies $\Sigma;\emptyset \vdash U : \psi' \to \phi$ and $\Sigma;\emptyset \vdash V : \psi'$ for some ϕ, ψ'. Therefore by Lemma 1(5) $U = \lambda x.A'$ and $\Sigma;\emptyset \vdash \lambda x.A' : \psi' \to \psi$ and so $\lambda x.A'$ is closed by Proposition 1. Rule (app) is applicable and $B = (x = V); A'$.

- Case $E \circ V \diamond B'$. By Lemma 2(10) $\Sigma;\emptyset \vdash E \circ V \diamond B' : \psi$ implies $\Sigma;\emptyset \vdash E : \tau$ and $\Sigma;\emptyset \vdash V : \langle \Gamma', \psi \rangle$ and $\Sigma;\emptyset \vdash B' : \phi$ for some ϕ, τ, Γ'. From Lemma 1(7) $V = \langle A' \rangle$. Therefore either rule (defOK) or rule (defEXC) is applicable.

- Case $E \frown U\ m(V)$. By Lemma 2(17) $\Sigma;\emptyset \vdash E \frown U\ m(V) : \psi$ implies $\Sigma;\emptyset \vdash U : \mathtt{ref}\tau$ for some $\tau = \mu\mathtt{t}.m^{\dagger\phi'}, \mathtt{lkp}^{!\phi}, \mathtt{ctx}^{!\varpi}, \tau'$. Therefore by Lemma 1(6) $U = \iota$ and $\iota : \tau \in \Sigma$. From $\Sigma \vdash \sigma$ we get $\Sigma;\emptyset \vdash \sigma(\iota) : \tau$, which implies $\sigma(\iota) = F \cdot (\mathtt{ctx}{=}E') \cdot (\mathtt{lkp}{=}V')$ for some F, E', V' by Lemma 1(3). (send) is applicable and $B = (\lambda b.(b)^{E'}\ \iota\ V)\ (V'\ E\ \iota\ \langle m \rangle)$.

- Case $E; \lambda x.A'$. We can always assume that x is not $\mathsf{FV}(E)$, by α-renaming. Therefore, rule (absS) is applicable.

- Case $E;x$. By Lemma 2(9) $\Sigma;\emptyset \vdash E;x : \psi$ implies $\Sigma;\emptyset \vdash E : \tau$ and $\Sigma;(\tau)^! \vdash x : \psi$. By Lemma 2(3) $x^{\psi'} \in \tau$ for some $\psi' \sqsubseteq \psi$. Therefore $\tau = x^{\psi'}, \tau'$ for some τ'. From Lemma 1(3) we derive that $E \equiv E'\cdot(x{=}V)$ for some E' and V. So rule (varS) is applicable, and $B = V$.

- Case $E;(x)^F$. The proof is similar to the case $E;x$.

Case: $\mathcal{C} \neq [\,]$. By Lemma 4(1) we have $\Sigma;\emptyset \vdash \mathcal{R} : \psi'$ for some ψ'. Then by previous case $\mathcal{R} \to_r A'$ for some A', and we conclude by applying rule (cont). $\square$