Alias and Union Types for Delegation

Christopher Anderson
Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, U.K.
cla97@doc.ic.ac.uk

Franco Barbanera
Dipartimento di Matematica e Informatica
Università di Catania, Viale A. Doria, 95125 Catania, Italy
barba@dmi.unict.it

Mariangiola Dezani-Ciancaglini
Dipartimento di Informatica
Università di Torino, C.so Svizzera, 158, 10149 Torino, Italy
dezani@di.unito.it

***Abstract*— We adapt the *aliasing constraints* approach for designing a flexible typing of evolving objects. Types are singleton types (addresses of objects, as a matter of fact) or finite "unions" of these. Their relevance is mainly due to the sort of *safety property* they guarantee. In particular we provide a type system for an imperative object based calculus with delegation and conditional expressions and which supports method and delegate overriding, addition, and removing.**

***Index Terms*— object based calculi, delegation, alias types, union types, effects.**

I. Introduction

In the global computing scenario it is crucial to develop software exhibiting three key properties: *evolution*, *incompleteness*, and *safety*. In the context of the object oriented paradigm the first two properties amount to have objects with method and delegate overriding, addition, removing and which can delegate executions of methods to other objects [8], [1], [4].

A traditional approach for ensuring safety is typing. There is a large literature about calculi of objects with types (see [3], [5], [9], and their references), where safety is interpreted mainly as the property that well typed programs cannot go wrong, i.e. that no *message not understood* exception can be thrown. This is also the approach to program safety of the present paper, but whereas many of the proposed type systems in the literature are for functional object calculi and some of them are for imperative object calculi, we focus on an imperative object calculus with method and delegate updating, removal and delegation. To our knowledge no typing has been proposed for calculi with these features.

Work partially supported by EU within the FET - Global Computing initiative, project DART ST-2001-33477, by MURST Cofin'01 projects COMETA and NAPOLI, and by MURST Cofin'02 project McTati. The funding bodies are not responsible for any use that might be made of the results presented here.

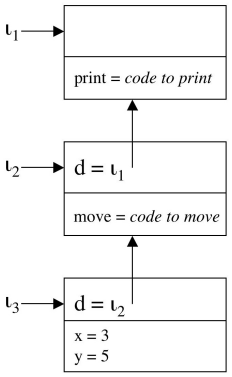

Fig. 1. Point with delegation

The aim of the present paper is then to partially fill the gap between the theory of types and the imperative object calculi with delegation. This is achieved by means of a simple idea: to adapt the *alias types* approach [10], [11] to the case of objects with delegation.

For low level code the *alias types* methodology has produced type systems which are collections of *aliasing constraints*. These constraints describe the shape of the store and every function uses them to specify the store that it expects. The pointers have *singleton types* which are the locations themselves.

Our proposal is to type expressions with finite unions

of singleton types, i.e. with finite sets of the logical and physical addresses. The intuition is that if an expression a is typed with a set of addresses then the evaluation of a if terminates produces an object whose address belongs to this set. The union is necessary in order to type conditional expressions.

The environments give type assumptions for term variables together with constraints on the (typed) sets of methods and delegates of the objects. The satisfaction of such constraints guarantees that a typable program can be *safely* evaluated. A key choice in the system is the typing of the method bodies: here the types give also complete information about the environments in which the bodies need to be typed. To correctly type a method call we require that the environment of the call represents (at least) the constraints needed to type the method body.

As a test case we apply this approach to δ^+, the extension with conditional expressions of the calculus δ for imperative object based delegation introduced in [4].

It is worth mentioning that recently the same approach has been applied to a calculus for "environment-aware" computation [6].

The present paper is organized as follows: in Section II we introduce δ^+. Section III presents the type assignment system: types, typing rules and soundness proof.

A preliminary version of the present paper is [2].

II. The Calculus

We present δ^+, a minimal imperative object based calculus with delegation. Delegation has also been studied in [7] and its derivatives. Also in [3] delegation is encoded into some of the variants of the ς-calculus. However, because the ς-calculus does not support the addition of methods and because the derivatives of [7] model delegation through copying, neither adequately reflect the situation where an object o_1 delegates to another object o_2, then o_2's method body for m is modified or added and subsequent method call of m on o_1 results in execution of the modified method body rather than the original one.

With delegation we can represent a point by *three* objects. One object that knows how to print points:

$$\iota_1.\mathsf{print} \lhd \mathit{PrintCode}$$

Here we have used lazy update, $\lhd$, to add the unevaluated code for printing to ι_1 with method identifier print. We abbreviate the body of method print by $\mathit{PrintCode}$. Similarly another object knows how to move objects:

$$\iota_2.\mathsf{move} \lhd (\mathsf{self}.\mathsf{x} \blacktriangleleft \mathsf{self}.\mathsf{x} + 1)$$

Again, lazy update is used to add method move to object ι_2. The body of method move uses eager update, $\blacktriangleleft$, to increment the x method of the receiver, identified by self. Thus, self.x + 1 is evaluated and the result stored in self.x. Note that for simplicity, we use the literals 1,2 ... as a shorthand for the object representations of the corresponding numbers.

Finally we have an object containing the x and y coordinates:

$$((\iota_3.\mathsf{x} \blacktriangleleft 3).\mathsf{y} \blacktriangleleft 5)$$

We now link the objects together using delegate update, $\mathsf{a}@\mathsf{d} \blacktriangleleft \mathsf{b}$:

$$\iota_3@\mathsf{d} \blacktriangleleft \iota_2; \iota_2@\mathsf{d} \blacktriangleleft \iota_1$$

Figure 1 shows the three objects representing a point at coordinates (3,5). The objects are represented in two parts: the upper part contains the delegates and the lower part contains the methods. When ι_3 receives a move message it delegates it to ι_2. Similarly, when ι_3 receives a print message it delegates it to ι_2 and ι_2 delegates it to ι_1. Thus, delegation allows sharing of methods between objects and thus the objects may be defined in terms of each other. Any changes to methods will be visible to both delegator and delegate. Hence, if we were to update the print code of ι_1 this would affect the behaviour of both ι_2 and ι_3.

The following example shows the difference between eager and lazy updating. The evaluation of the eager updating $\iota.\mathsf{m} \blacktriangleleft (\iota'.\mathsf{m}' \lhd \mathsf{self})$ first evaluates the body of the method, i.e. $\iota'.\mathsf{m}' \lhd \mathsf{self}$, and then updates the m method of the object at ι with the result of the body evaluation, i.e. ι'.

Then for all stores σ

$$\iota.\mathsf{m} \blacktriangleleft (\iota'.\mathsf{m}' \lhd \mathsf{self})\ ;\ \iota'.\mathsf{m}', \sigma \ \leadsto_{\delta}\ \iota', \sigma'$$

where σ' is obtained from σ by updating the method m in the object at ι and the method m' in the object at ι'.

If in the previous expression we change the method updating from eager to lazy, and the object at ι' and its delegates do not have the method m', we get a `stuckErr`:

$$\iota.\mathsf{m} \lhd (\iota'.\mathsf{m}' \lhd \mathsf{self})\ ;\ \iota'.\mathsf{m}', \sigma \ \leadsto_{\delta}\ \texttt{stuckErr}, \sigma'$$

where σ' is obtained from σ by updating only the method m in the object at ι with the (unevaluated) body $\iota'.\mathsf{m}' \lhd \mathsf{self}$.

Consider the following example which demonstrates an object with two delegates:

$$\iota@\mathsf{d}_1 \blacktriangleleft \iota'; \iota@\mathsf{d}_2 \blacktriangleleft \iota''; \iota'.\mathsf{m} \lhd (\iota''.\mathsf{m}' \blacktriangleleft \mathsf{self}); \iota.\mathsf{m}; \iota.\mathsf{m}'$$

We assume that in the initial state the object ι and its delegates do not have the methods m and m'. Firstly, ι' and ι'' are made delegates of ι, followed by the lazy addition of method m to ι'. When $\iota.\mathsf{m}$ is executed, ι delegates execution of m to ι' which adds a new method m' to ι'' with body ι. When $\iota.\mathsf{m}'$ is executed ι delegates

$$
\begin{array}{rcll}
\iota & \in & \mathsf{PhysAddress} & \\
\mathsf{m} & \in & \mathsf{MethID} & \\
\mathsf{d} & \in & \mathsf{DelID} & \\
 & & \mathsf{MethID} \cap \mathsf{DelID} = \emptyset & \\
 & & & \\
\mathsf{a},\mathsf{b},\mathsf{c} & \in & \mathsf{Exp} \quad ::= & \\
 & & \iota & \text{physical address} \\
 & & \mathsf{a.m} & \text{method invocation} \\
 & & \mathsf{a.m} \blacktriangleleft \mathsf{b} & \text{eager update} \\
 & & \mathsf{a.m} \lhd \mathsf{b} & \text{lazy update} \\
 & & \mathsf{clone(a)} & \text{clone (shallow)} \\
 & & \mathsf{self} & \text{receiver} \\
 & & \mathsf{a} \rhd_{\bullet} \mathsf{m} & \text{method remove} \\
 & & \mathsf{a@d.m} & \text{delegate invocation} \\
 & & \mathsf{a@d} \blacktriangleleft \mathsf{b} & \text{delegate update} \\
 & & \mathsf{a} \rhd_{@} \mathsf{d} & \text{delegate remove} \\
 & & \mathsf{if\ a\ then\ b\ else\ c} & \text{conditional} \\
 & & \mathsf{a\ ;\ b} & \text{composition}
\end{array}
$$

Fig. 2. Syntax of δ^+

execution of m' to ι''. Therefore if ι and its delegates do not have the methods m and m' in σ

$$\iota@\mathsf{d}_1 \blacktriangleleft \iota'; \iota@\mathsf{d}_2 \blacktriangleleft \iota''; \iota'.\mathsf{m} \lhd (\iota''.\mathsf{m}' \blacktriangleleft \mathsf{self}); \iota.\mathsf{m}; \iota.\mathsf{m}', \sigma \rightsquigarrow_{\delta} \iota, \sigma'$$

where σ' is obtained from σ by updating the delegates $\mathsf{d}_1 = \iota'$ and $\mathsf{d}_2 = \iota''$ in the object at ι, by adding the method m with body $\iota''.\mathsf{m}' \blacktriangleleft \mathsf{self}$ in the object at ι' and by adding the method m' with body ι in the object at ι''.

If instead we had written:

$$\iota@\mathsf{d}_1 \blacktriangleleft \iota'; \iota@\mathsf{d}_2 \blacktriangleleft \iota''; \iota'.\mathsf{m} \lhd (\iota''.\mathsf{m} \blacktriangleleft \mathsf{self}); \iota.\mathsf{m}; \iota.\mathsf{m}$$

after the first call to $\iota.\mathsf{m}$ both ι' and ι'' have a method m. Therefore, the second call to $\iota.\mathsf{m}$ is ambiguous and execution will produce `stuckErr`. However, we can write $\iota@\mathsf{d}_2.\mathsf{m}$ to execute m in ι'' with the receiver being ι.

A. Syntax

The syntax (shown in Figure 2) defines twelve kinds of expressions: physical addresses, method invocation, eager and lazy update, clone, self, method removal, delegate invocation/update/removal, conditional and composition. PhysAddress, $\iota_{\mathsf{f}}, \iota_1, \ldots \iota_{\mathsf{n}}$, is the set of physical addresses: they play, in some sense, the role of variables. Method and delegate identifiers are taken from the disjoint infinite sets of names MethID and DelID respectively. For what concerns conditional expression, instead of explicitly introducing constants for boolean values, we shall treat a particular address, ι_{f}, as false and all other addresses as true. This choice both prevents the calculus from growing too cumbersome for our foundational aims and conforms to the implementations of many actual programming languages.

We convene that $\mathsf{n} \in \mathsf{MethID} \uplus \mathsf{DelID}$, where $\uplus$ denotes union of disjoint sets and we use $(\mathsf{n}{=}\mathsf{b}) \in \mathsf{o}$ as short for:
the object o has

- the method identifier n with associated body b if $\mathsf{n} \in \mathsf{MethID}$ or
- the delegate identifier n with associated physical address b if $\mathsf{n} \in \mathsf{DelID}$.

B. Semantics

The operational semantics for δ^+ is given in Figure 3, but for the rules of stuck error propagation, which are standard. It rewrites pairs of expression and stores into pairs of physical addresses or `stuckErr` and stores.

By $\rightharpoonup_{\mathsf{fin}}$ we denote finite mappings. The stores map physical addresses to objects and self to a memory address. Objects are finite mappings from method names to expressions and from delegate names to physical addresses.

$$
\begin{array}{rcl}
\rightsquigarrow_{\delta} & : & \mathsf{Exp} \times \mathsf{Store} \rightharpoonup_{\mathsf{fin}} \mathsf{PhysAddress} \times \mathsf{Store} \\
\mathsf{Store} & = & (\{\mathsf{self}\} \to \mathsf{PhysAddress}) \cup (\mathsf{PhysAddress} \rightharpoonup_{\mathsf{fin}} \mathsf{Obj}) \\
\mathsf{Obj} & = & (\mathsf{MethID} \rightharpoonup_{\mathsf{fin}} \mathsf{Exp}) \cup (\mathsf{DelID} \rightharpoonup_{\mathsf{fin}} \mathsf{PhysAddress})
\end{array}
$$

Let $\mathcal{U}df$ denote undefined. The rewrite rules (*Select*) and (*Delegate Select*) use lookup functions $\mathcal{L}ook$ and $\mathcal{L}ook'$ shown in Figure 4. $\mathcal{L}ook'$ returns the set of pairs of addresses and bodies corresponding to a method identifier and an address in a given store. Lookup starts

(*Self*)

$$\frac{}{\mathsf{self}, \sigma \ \leadsto_\delta \ \sigma(\mathsf{self}), \sigma}$$

(*Addr*)

$$\frac{}{\iota, \sigma \ \leadsto_\delta \ \iota, \sigma}$$

(*Select*)

$$\frac{\mathsf{a}, \sigma \ \leadsto_\delta \ \iota, \sigma' \quad \mathcal{L}ook(\sigma', \iota, \mathsf{m}) = \mathsf{b} \quad \sigma''' = \sigma'[\mathsf{self} \mapsto \iota] \quad \mathsf{b}, \sigma''' \ \leadsto_\delta \ \iota', \sigma''}{\mathsf{a.m}, \sigma \ \leadsto_\delta \ \iota', \sigma''[\mathsf{self} \mapsto \sigma(\mathsf{self})]}$$

(*Delegate Select*)

$$\frac{\mathsf{a}, \sigma \ \leadsto_\delta \ \iota, \sigma' \quad \mathcal{L}ook(\sigma', \iota, \mathsf{d}) = \iota' \qquad \mathcal{L}ook(\sigma', \iota', \mathsf{m}) = \mathsf{b} \quad \sigma''' = \sigma'[\mathsf{self} \mapsto \iota] \quad \mathsf{b}, \sigma''' \ \leadsto_\delta \ \iota', \sigma''}{\mathsf{a@d.m}, \sigma \ \leadsto_\delta \ \iota', \sigma''[\mathsf{self} \mapsto \sigma(\mathsf{self})]}$$

(*Lazy Update*)

$$\frac{\mathsf{a}, \sigma \ \leadsto_\delta \ \iota, \sigma'}{\mathsf{a.m} \lhd \mathsf{b}, \sigma \ \leadsto_\delta \ \iota, \sigma'\{\iota.\mathsf{m} \lhd^+ \mathsf{b}\}}$$

(*Eager Update*)

$$\frac{\mathsf{a}, \sigma \ \leadsto_\delta \ \iota, \sigma' \qquad \mathsf{b}, \sigma' \ \leadsto_\delta \ \iota', \sigma''}{\mathsf{a.m} \blacktriangleleft \mathsf{b}, \sigma \ \leadsto_\delta \ \iota, \sigma''\{\iota.\mathsf{m} \lhd^+ \iota'\}}$$

(*Delegate Update*)

$$\frac{\mathsf{a}, \sigma \ \leadsto_\delta \ \iota, \sigma' \quad \mathsf{b}, \sigma' \ \leadsto_\delta \ \iota', \sigma''}{\mathsf{a@d} \blacktriangleleft \mathsf{b}, \sigma \ \leadsto_\delta \ \iota, \sigma''\{\iota.\mathsf{d} \lhd^+ \iota'\}}$$

(*Clone*)

$$\frac{\mathsf{a}, \sigma \ \leadsto_\delta \ \iota, \sigma' \qquad \iota' \notin dom(\sigma') \quad \sigma'' = \sigma'[\iota' \mapsto \sigma'(\iota)]}{\mathsf{clone(a)}, \sigma \ \leadsto_\delta \ \iota', \sigma''}$$

(*Method Remove*)

$$\frac{\mathsf{a}, \sigma \ \leadsto_\delta \ \iota, \sigma'}{\mathsf{a} \rhd_\bullet \mathsf{m}, \sigma \ \leadsto_\delta \ \iota, \sigma'\{\iota \rhd \mathsf{m}\}}$$

(*Delegate Remove*)

$$\frac{\mathsf{a}, \sigma \ \leadsto_\delta \ \iota, \sigma'}{\mathsf{a} \rhd_@ \mathsf{d}, \sigma \ \leadsto_\delta \ \iota, \sigma'\{\iota \rhd \mathsf{d}\}}$$

(*Composition*)

$$\frac{\mathsf{a}, \sigma \ \leadsto_\delta \ \iota, \sigma' \quad \mathsf{b}, \sigma' \ \leadsto_\delta \ \iota', \sigma''}{\mathsf{a} \ ; \mathsf{b}, \sigma \ \leadsto_\delta \ \iota', \sigma''}$$

(*Stuck Select*)

$$\frac{\mathsf{a}, \sigma \ \leadsto_\delta \ \iota, \sigma' \quad \mathcal{L}ook(\sigma', \iota, \mathsf{m}) = \mathcal{U}df}{\mathsf{a.m}, \sigma \ \leadsto_\delta \ \mathtt{stuckErr}, \sigma'}$$

(*Stuck1 Delegate Select*)

$$\frac{\mathsf{a}, \sigma \ \leadsto_\delta \ \iota, \sigma' \quad \mathcal{L}ook(\sigma', \iota, \mathsf{d}) = \mathcal{U}df}{\mathsf{a@d.m}, \sigma \ \leadsto_\delta \ \mathtt{stuckErr}, \sigma'}$$

(*Stuck2 Delegate Select*)

$$\frac{\mathsf{a}, \sigma \ \leadsto_\delta \ \iota, \sigma' \quad \mathcal{L}ook(\sigma', \iota, \mathsf{d}) = \iota' \quad \mathcal{L}ook(\sigma', \iota', \mathsf{m}) = \mathcal{U}df}{\mathsf{a@d.m}, \sigma \ \leadsto_\delta \ \mathtt{stuckErr}, \sigma'}$$

(*Conditional true*)

$$\frac{\mathsf{b}, \sigma \ \leadsto_\delta \ \iota', \sigma \qquad \iota' \neq \iota_\mathsf{f} \quad \mathsf{a}, \sigma \ \leadsto_\delta \ \iota, \sigma'}{\mathsf{if\ b\ then\ a\ else\ a'}, \sigma \ \leadsto_\delta \ \iota, \sigma'}$$

(*Conditional false*)

$$\frac{\mathsf{b}, \sigma \ \leadsto_\delta \ \iota_\mathsf{f}, \sigma \quad \mathsf{a}', \sigma \ \leadsto_\delta \ \iota, \sigma'}{\mathsf{if\ b\ then\ a\ else\ a'}, \sigma \ \leadsto_\delta \ \iota, \sigma'}$$

(*Conditional stuck*)

$$\frac{\mathsf{b}, \sigma \ \leadsto_\delta \ \iota, \sigma' \qquad \sigma \neq \sigma'}{\mathsf{if\ b\ then\ a\ else\ a'}, \sigma \ \leadsto_\delta \ \mathtt{stuckErr}, \sigma'}$$

Fig. 3. Operational Semantics of δ^+

in object $\sigma(\iota)$, and if no method body is found, then the search continues in the delegates. Note that $\mathcal{L}ook$ is defined only if $\mathcal{L}ook'$ finds *exactly* one method or delegate body. So, if an object has several method bodies in several different delegates, or if no method body can be found in the object or its delegates, then

evaluation produces a stuck error (rules (*Stuck Select*), (*Stuck1 Delegate Select*), and (*Stuck2 Delegate Select*)). If instead a unique method body is found, this body is evaluated in the context of a store where self is bound to the address of the receiver. Finally, self is set back to the address it had before the method invocation.

There are two kinds of update: eager and lazy, whose operational behaviour is described by rules (*Eager Update*) and (*Lazy Update*) respectively. They differ in their treatment of the new body b. Lazy update replaces the method body identified by m with the unevaluated body b. In eager update the body b is evaluated before the update occurs. Hence, lazy update is like method update and eager update is like field update. Both updates use $\mathcal{L}ook$ and $\mathcal{L}ook'$ to check if there is *exactly* one object containing the specified method or delegate, starting the lookup from the object which receives the message. If such an object is found, the update is realized by overwriting the body of the method (or the physical address of the delegate). Otherwise the object which receives the message is extended by the method (or the delegate). This is done by the store update, $\sigma\{\iota.\mathsf{n} \lhd^{+} \mathsf{b}\}$ defined in Figure 5.[1]

Delegate update (*Delegate Update*) adds or updates the delegate identified by d in the receiver a with the evaluated body b.

The rule for cloning, (*Clone*), evaluates the object a then it allocates a new address, copies the object to the new address and returns the new address.

We define:

$$\sigma[\iota \mapsto \mathsf{o}](\iota') = \begin{cases} \mathsf{o} & \text{if } \iota' = \iota, \\ \sigma(\iota') & \text{otherwise.} \end{cases}$$

Rules for method and delegate removal, (*Method Remove*) and (*Delegate Remove*), first evaluate the receiver then return the receiver with the delegate or method removed. The operation of store removal, $\sigma\{\iota \rhd \mathsf{n}\}$, is defined in Figure 5.

The reduction rules for the conditionals, (*Conditional false*) and (*Conditional true*), are quite standard, but for the requirement that the evaluation of the condition b must leave the state unchanged. This restriction does not affect the expressive power of the calculus and it is however handy if we wish to have a sound type system for it (Theorem 1).

III. The Type Assignment System

Looking at the operational semantics of δ^{+} one easily sees that a `stuckErr` is generated when:

- a method invocation or a delegate invocation does not find a method or a delegate;
- the evaluation of the condition in a conditional expression changes the state.

[1]We slightly changed the definition of storage update in [4] to avoid to have operations which modify more than one object.

a, b	$\in$	Exp	::=	x term variable
				ι physical address
				:
				as in Figure 2

Fig. 6. Syntax of δ^{++}

To assure that well typed expressions cannot go wrong we need a type system tracing for all objects how methods and delegates are added, updated and removed. We get this simply by allowing types of objects to be their (logical or physical) addresses or a finite sets of these. Finite sets shall be represented by means of a "union" operator on object types. The necessity of introducing a union operator is due to the presence of conditional expressions, since we cannot know *a priori* which branch of the conditional will be evaluated, and this problem reflects on the typing as well. The introduction of the union operator, however, needs to be coupled with the introduction of term variables. In fact our typing rules concerning conditional expression will mimic the logical rules in natural deduction for the introduction and elimination of the disjunction (see Rules (R-union-I) and (R-union-E) in Figure 7). The syntax of our calculus needs then to be extended with term variables. We stress that such variables are needed only for typing motivations and indeed they possibly appear inside derivations but not in the conclusions of typings for δ^{+} terms. Then, if term variables are ranged over by x, y,..., the "variable-extended" syntax of our calculus is as in Figure 6. We call δ^{++} this calculus.

The typing judgements of our system are of the following shape:

$$\Gamma \mid \Delta \vdash \mathsf{a} : \tau, \varphi$$

where:

- the address environment Γ gives information about the types of delegates and methods of objects at fixed addresses. In this way the environment Γ represents the constraints the store must satisfy in order to successfully evaluate the expression a;
- the term environment Δ gives information about address environments, object types and effects of the term variables present in a. These variables have to be replaced, using Rule (R-union-E), by "ground" terms (which, in well-typed terms, will

$$\begin{array}{rcl} \mathcal{L}ook & :: & \mathsf{Store} \times \mathsf{PhysAddress} \times (\mathsf{MethID} \uplus \mathsf{DelID}) \to \mathsf{Exp} \uplus \{\mathcal{U}df\} \\ \mathcal{L}ook' & :: & \mathsf{Store} \times \mathsf{PhysAddress} \times (\mathsf{MethID} \uplus \mathsf{DelID}) \to \mathcal{P}(\mathsf{Exp} \times \mathsf{PhysAddress}) \end{array}$$

$$\mathcal{L}ook(\sigma,\iota,\mathsf{n}) \;=\; \begin{cases} \mathsf{b} & \text{if } \mathcal{L}ook'(\sigma,\iota,\mathsf{n}) = \{(\mathsf{b},\iota')\} \\ \mathcal{U}df & \text{otherwise} \end{cases}$$

$$\mathcal{L}ook'(\sigma,\iota,\mathsf{n}) \;=\; \begin{cases} \{(\mathsf{b},\iota)\} & \text{if } (\mathsf{n}=\mathsf{b}) \in \sigma(\iota) \\ \bigcup\limits_{\iota' \in I} \mathcal{L}ook'(\sigma,\iota',\mathsf{n}) & \text{otherwise} \end{cases}$$
where $I = \{\iota' \mid (\mathsf{d}=\iota') \in \sigma(\iota) \text{ for some } \mathsf{d}\}$

Fig. 4. The $\mathcal{L}ook$ and $\mathcal{L}ook'$ Functions

$$\sigma\{\iota.\mathsf{n} \lhd^{+} \mathsf{b}\}(\iota')(\mathsf{n}') \;=\; \begin{cases} \mathsf{b} & \text{if } \mathsf{n}'=\mathsf{n}, \iota'=\iota \text{ and } \mathcal{L}ook(\sigma,\iota,\mathsf{n}) = \mathcal{U}df \\ \mathsf{b} & \text{if } \mathsf{n}=\mathsf{n}' \text{ and } \mathcal{L}ook'(\sigma,\iota,\mathsf{n}) = \{(\mathsf{b}',\iota')\} \\ \sigma(\iota')(\mathsf{n}') & \text{otherwise} \end{cases}$$

$$\sigma\{\iota \rhd \mathsf{n}\}(\iota')(\mathsf{n}') \;=\; \begin{cases} \mathcal{U}df & \text{if } \mathsf{n}'=\mathsf{n} \text{ and } \iota'=\iota \\ \sigma(\iota')(\mathsf{n}') & \text{otherwise} \end{cases}$$

Fig. 5. Store Update and Remove

be conditional expressions) in order to get type derivations for expressions in δ^{+};
- the type τ gives a set of possible addresses of the object, which is the value (if it exists) of the expression a after replacing term variables with δ^{+} expressions;
- the effect φ gives the changes of the address environment due to the typing of a in Γ, in this way the effect φ represents the changes of the store due to the evaluation of a in a store satisfying Γ.

In other words the evaluation of a, whenever it terminates, is guaranteed to produce an object whose address belongs to the set described by τ and a store satisfying the constraints $\varphi(\Gamma)$, if the evaluation starts with a store satisfying Γ.

Typing a delegate update gives:

$$\emptyset \mid \emptyset \vdash \iota@\mathsf{d} \blacktriangleleft \iota' : \mathsf{Obj}(\iota), \varphi$$

where effect

$\varphi = \{[\iota : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle]\} \circ \{[\iota' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle]\} \circ \{\{\iota : \langle\!\langle \mathsf{d} : \mathsf{Obj}(\iota') \rangle\!\rangle_{@}\}\}$

says that ι, ι' are empty objects and ι' is the delegate d at ι.

A more interesting example deals with the eager and lazy method update:

$$\emptyset \mid \emptyset \vdash \iota'.\mathsf{m} \lhd (\iota''.\mathsf{m}' \blacktriangleleft \mathsf{self}) : \mathsf{Obj}(\iota'), \varphi'$$

where

$\varphi' = \{[\iota' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle]\} \circ \{\{\iota' : \langle\!\langle \mathsf{m} : \langle \emptyset \mid \emptyset, \mathsf{Obj}(\iota''), \varphi'' \rangle \rangle\!\rangle_{\bullet}\}\}$

and

$\varphi'' = \{[\iota'' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle]\} \circ \{\{\iota'' : \langle\!\langle \mathsf{m}' : \mathsf{Obj}(\iota_{\mathsf{self}}) \rangle\!\rangle_{\bullet}\}\}.$

We obtain an effect containing a type which, in turn, contains an effect.
The effect φ' says that ι' is an empty object and the method m is updated at ι' with a body of type $\langle \emptyset \mid \emptyset, \mathsf{Obj}(\iota'), \varphi'' \rangle$. The effect φ'' inside the type of (the body of) m takes into account that the evaluation of this body will assign the empty row to ι'' and will update the method m' at ι'' with type $\mathsf{Obj}(\iota_{\mathsf{self}})$.

A. Types, Environments, Effects, Judgments

To define types we need to consider, besides the set of physical addresses (PhysAddress, ranged over by

$$\frac{}{\Gamma \mid \Delta; x : \Gamma, \tau, \varphi \vdash x : \tau, \varphi}\ \text{(Ax-}x\text{-init)} \qquad \frac{\iota \notin \Gamma}{\Gamma \mid \Delta \vdash \iota : \mathsf{Obj}(\iota), \{[\iota : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle]\}}\ \text{(Ax-}\iota\text{-init)}$$

$$\frac{\iota \in \Gamma}{\Gamma \mid \Delta \vdash \iota : \mathsf{Obj}(\iota), \mathsf{id}}\ \text{(Ax-}\iota\text{)} \qquad \frac{\iota_{\mathsf{self}} \in \Gamma}{\Gamma \mid \Delta \vdash \mathsf{self} : \mathsf{Obj}(\iota_{\mathsf{self}}), \mathsf{id}}\ \text{(Ax-self)}$$

$$\frac{\Gamma \mid \Delta \vdash \mathsf{a} : \mathsf{Obj}(\zeta), \varphi \quad \varphi(\Gamma)(\zeta) = \rho \quad \iota_{\mathsf{c}} \notin \varphi(\Gamma)}{\Gamma \mid \Delta \vdash \mathsf{clone(a)} : \mathsf{Obj}(\iota_{\mathsf{c}}), \varphi \circ \{[\iota_{\mathsf{c}} : \rho]\}}\ \text{(R-clone)} \qquad \frac{\Gamma \mid \Delta \vdash \mathsf{a} : \tau', \varphi \quad \varphi(\Gamma) \mid \Delta \vdash \mathsf{b} : \tau, \varphi'}{\Gamma \mid \Delta \vdash \mathsf{a\ ;\ b} : \tau, \varphi \circ \varphi'}\ \text{(R-comp)}$$

$$\frac{\Gamma \mid \Delta \vdash \mathsf{a} : \mathsf{Obj}(\zeta), \varphi \quad \mathcal{L}t(\varphi(\Gamma), \zeta, \mathsf{m}) = \tau}{\Gamma \mid \Delta \vdash \mathsf{a.m} : \tau[\zeta/\iota_{\mathsf{self}}], \varphi}\ \text{(R-eager-sel)} \qquad \frac{\Gamma \mid \Delta \vdash \mathsf{a} : \mathsf{Obj}(\zeta), \varphi \quad \mathcal{L}t(\varphi(\Gamma), \zeta, \mathsf{d}) = \mathsf{Obj}(\zeta') \quad \mathcal{L}t(\varphi(\Gamma), \zeta', \mathsf{m}) = \tau}{\Gamma \mid \Delta \vdash \mathsf{a@d.m} : \tau[\zeta/\iota_{\mathsf{self}}], \varphi}\ \text{(R-del-eager-sel)}$$

$$\frac{\Gamma \mid \Delta \vdash \mathsf{a} : \mathsf{Obj}(\zeta), \varphi \qquad \mathcal{L}t(\varphi(\Gamma), \zeta, \mathsf{m}) = \langle \Gamma' \mid \Delta', \tau, \varphi' \rangle \quad \varphi(\Gamma)(\zeta) = \rho \quad \varphi \circ \{[\iota_{\mathsf{self}} : \rho]\}(\Gamma) \leq \Gamma' \quad \Delta' \subseteq \Delta \qquad \{\iota_{\mathsf{c}} \mid \iota_{\mathsf{c}} \in \varphi'(\Gamma') \ \&\ \iota_{\mathsf{c}} \in \varphi(\Gamma) \ \&\ \iota_{\mathsf{c}} \notin \Gamma'\} = \emptyset}{\Gamma \mid \Delta \vdash \mathsf{a.m} : \tau[\zeta/\iota_{\mathsf{self}}], \varphi \circ \varphi'[\zeta/\iota_{\mathsf{self}}]}\ \text{(R-lazy-sel)}$$

$$\frac{\Gamma \mid \Delta \vdash \mathsf{a} : \mathsf{Obj}(\zeta), \varphi \qquad \mathcal{L}t(\varphi(\Gamma), \zeta, \mathsf{d}) = \mathsf{Obj}(\zeta') \quad \mathcal{L}t(\varphi(\Gamma), \zeta', \mathsf{m}) = \langle \Gamma' \mid \Delta', \tau, \varphi' \rangle \quad \varphi(\Gamma)(\zeta) = \rho \quad \varphi \circ \{[\iota_{\mathsf{self}} : \rho]\}(\Gamma) \leq \Gamma' \quad \Delta' \subseteq \Delta \qquad \{\iota_{\mathsf{c}} \mid \iota_{\mathsf{c}} \in \varphi'(\Gamma') \ \&\ \iota_{\mathsf{c}} \in \varphi(\Gamma) \ \&\ \iota_{\mathsf{c}} \notin \Gamma'\} = \emptyset}{\Gamma \mid \Delta \vdash \mathsf{a@d.m} : \tau[\zeta/\iota_{\mathsf{self}}], \varphi \circ \varphi'[\zeta/\iota_{\mathsf{self}}]}\ \text{(R-del-lazy-sel)}$$

$$\frac{\Gamma \mid \Delta \vdash \mathsf{a} : \mathsf{Obj}(\zeta), \varphi \quad \mathcal{L}a(\varphi(\Gamma), \zeta, \mathsf{m}) = \zeta' \qquad \varphi(\Gamma)(\zeta') = \rho \quad \varphi \circ \{[\iota_{\mathsf{self}} : \rho]\}(\Gamma) = \Gamma' \quad \Gamma' \mid \Delta \vdash \mathsf{b} : \tau, \varphi'}{\Gamma \mid \Delta \vdash \mathsf{a.m} \blacktriangleleft \mathsf{b} : \mathsf{Obj}(\zeta), \varphi \circ \varphi' \circ \{\{\zeta' : \langle\!\langle \mathsf{m} : \tau \rangle\!\rangle_{\bullet}\}\}}\ \text{(R-eager-up)}$$

$$\frac{\Gamma \mid \Delta \vdash \mathsf{a} : \mathsf{Obj}(\zeta), \varphi \quad \mathcal{L}a(\varphi(\Gamma), \zeta, \mathsf{m}) = \zeta' \quad \Theta \vdash \mathsf{b} : \tau, \varphi'}{\Gamma \mid \Delta \vdash \mathsf{a.m} \lhd \mathsf{b} : \mathsf{Obj}(\zeta), \varphi \circ \{\{\zeta' : \langle\!\langle \mathsf{m} : \langle \Theta, \tau, \varphi' \rangle \rangle\!\rangle_{\bullet}\}\}}\ \text{(R-lazy-up)}$$

$$\frac{\Gamma \mid \Delta \vdash \mathsf{a} : \mathsf{Obj}(\zeta), \varphi \quad \varphi(\Gamma) \mid \Delta \vdash \mathsf{b} : \mathsf{Obj}(\zeta'), \varphi'}{\Gamma \mid \Delta \vdash \mathsf{a@d} \blacktriangleleft \mathsf{b} : \mathsf{Obj}(\zeta), \varphi \circ \varphi' \circ \{\{\zeta : \langle\!\langle \mathsf{d} : \mathsf{Obj}(\zeta') \rangle\!\rangle_{@}\}\}}\ \text{(R-del-up)}$$

$$\frac{\Gamma \mid \Delta \vdash \mathsf{a} : \mathsf{Obj}(\zeta), \varphi}{\Gamma \mid \Delta \vdash \mathsf{a} \rhd_{\bullet} \mathsf{m} : \mathsf{Obj}(\zeta), \varphi \circ \{\{\zeta : /_{\bullet}\mathsf{m}\}\}}\ \text{(R-met-rem)} \qquad \frac{\Gamma \mid \Delta \vdash \mathsf{a} : \mathsf{Obj}(\zeta), \varphi}{\Gamma \mid \Delta \vdash \mathsf{a} \rhd_{@} \mathsf{d} : \mathsf{Obj}(\zeta), \varphi \circ \{\{\zeta : /_{@}\mathsf{d}\}\}}\ \text{(R-del-rem)}$$

$$\frac{\Gamma \mid \Delta \vdash \mathsf{b} : \mathsf{Obj}(\zeta), \mathsf{id} \quad \Gamma \mid \Delta \vdash \mathsf{a} : \tau, \varphi \qquad \Gamma \mid \Delta \vdash \mathsf{a}' : \tau', \varphi}{\Gamma \mid \Delta \vdash \mathsf{if\ b\ then\ a\ else\ a}' : \tau \vee \tau', \varphi}\ \text{(R-union-I)}$$

$$\frac{\Gamma \mid \Delta; x : \Gamma', \tau', \varphi' \vdash \mathsf{c} : \tau, \varphi \quad \Gamma \mid \Delta; x : \Gamma', \tau'', \varphi' \vdash \mathsf{c} : \tau, \varphi \quad \Gamma' \mid \Delta \vdash \mathsf{b} : \mathsf{Obj}(\zeta), \mathsf{id} \quad \Gamma' \mid \Delta \vdash \mathsf{a} : \tau', \varphi' \qquad \Gamma' \mid \Delta \vdash \mathsf{a}' : \tau'', \varphi'}{\Gamma \mid \Delta \vdash \mathsf{c}[\mathsf{if\ b\ then\ a\ else\ a}'/x] : \tau, \varphi}\ \text{(R-union-E)}$$

$$\frac{\Gamma \mid \Delta \vdash \mathsf{b} : \mathsf{Obj}(\iota), \mathsf{id} \qquad \iota \neq \iota_{\mathsf{f}} \quad \Gamma \mid \Delta \vdash \mathsf{a} : \tau, \varphi \qquad \Gamma \mid \Delta \vdash \mathsf{a}' : \tau', \varphi'}{\Gamma \mid \Delta \vdash \mathsf{if\ b\ then\ a\ else\ a}' : \tau, \varphi}\ \text{(R-cond-t)}$$

$$\frac{\Gamma \mid \Delta \vdash \mathsf{b} : \mathsf{Obj}(\iota_{\mathsf{f}}), \mathsf{id} \quad \Gamma \mid \Delta \vdash \mathsf{a} : \tau, \varphi \qquad \Gamma \mid \Delta \vdash \mathsf{a}' : \tau', \varphi'}{\Gamma \mid \Delta \vdash \mathsf{if\ b\ then\ a\ else\ a}' : \tau', \varphi'}\ \text{(R-cond-f)}$$

Fig. 7. Typing Rules

OBJECT TYPES : τ	$\tau ::= \mathsf{Obj}(\zeta) \mid \tau \vee \tau$	where $\zeta \in \mathsf{PhysAddress} \uplus \mathsf{LogAddress}$
METHOD TYPES : λ	$\lambda ::= \tau \mid \langle \Theta, \tau, \varphi \rangle$	
METHOD ROWS : μ	$\mu ::= \langle\!\langle\, \rangle\!\rangle \mid \langle\!\langle \mu \mid \mathsf{m} : \lambda \rangle\!\rangle$	where $\mathsf{m} \in \mathsf{MethID}$
DELEGATE ROWS : ν	$\nu ::= \langle\!\langle\, \rangle\!\rangle \mid \langle\!\langle \nu \mid \mathsf{d} : \mathsf{Obj}(\zeta) \rangle\!\rangle$	where $\mathsf{d} \in \mathsf{DelID}$
ROW TYPES : ρ	$\rho ::= \nu \parallel \mu$	
ADDRESS ENVIRONMENTS : Γ	$\Gamma ::= \{\,\} \mid \Gamma \cup \{\zeta : \rho\}$	
TERM ENVIRONMENTS : Δ	$\Delta ::= \{\,\} \mid \Delta \cup \{x : \Gamma, \tau, \varphi\}$	
ENVIRONMENTS : Θ	$\Theta ::= \Gamma \mid \Delta$	
ROW OPERATORS : ψ	$\psi := /_\bullet \mathsf{m} \mid /_@ \mathsf{d} \mid \langle\!\langle \mathsf{m} : \lambda \rangle\!\rangle_\bullet \mid \langle\!\langle \mathsf{d} : \mathsf{Obj}(\zeta) \rangle\!\rangle_@$	
EFFECTS : φ	$\varphi ::= \mathsf{id} \mid \{\![\zeta : \rho]\!\} \mid \{\{\zeta : \psi\}\} \mid \varphi \circ \varphi$	

Fig. 8. Types, Environments, Effects

$$
\begin{array}{rcll}
/_\bullet \mathsf{m}(\nu \parallel \mu) & = & \nu \parallel \langle\!\langle \mathsf{m}' : \lambda \mid \mathsf{m}' : \lambda \in \mu \;\&\; \mathsf{m}' \neq \mathsf{m} \rangle\!\rangle & \text{method deletion} \\
/_@ \mathsf{d}(\nu \parallel \mu) & = & \langle\!\langle \mathsf{d}' : \mathsf{Obj}(\zeta) \mid \mathsf{d}' : \mathsf{Obj}(\zeta) \in \nu \;\&\; \mathsf{d}' \neq \mathsf{d} \rangle\!\rangle \parallel \mu & \text{delegate deletion} \\
\langle\!\langle \mathsf{m} : \lambda \rangle\!\rangle_\bullet(\nu \parallel \mu) & = & \nu \parallel \langle\!\langle /_\bullet \mathsf{m}(\mu) \mid \mathsf{m} : \lambda \rangle\!\rangle & \text{method update} \\
\langle\!\langle \mathsf{d} : \mathsf{Obj}(\zeta) \rangle\!\rangle_@(\nu \parallel \mu) & = & \langle\!\langle /_@ \mathsf{d}(\nu) \mid \mathsf{d} : \mathsf{Obj}(\zeta) \rangle\!\rangle \parallel \mu & \text{delegate update}
\end{array}
$$

Fig. 9. Row Operators

$$
\begin{array}{lcll}
\mathsf{id}(\Gamma) & = & \Gamma & \text{identity} \\
\{\![\zeta : \rho]\!\}(\Gamma) & = & \begin{cases} \Gamma & \text{if } \zeta \in \Gamma \;\&\; \zeta \neq \iota_{\mathsf{self}} \\ \{\zeta' : \rho \mid \zeta' : \rho \in \Gamma \;\&\; \zeta' \neq \zeta\} \cup \{\zeta : \rho\} & \text{otherwise} \end{cases} & \text{independent row updating} \\
\{\{\zeta : \psi\}\}(\Gamma) & = & \{\zeta' : \rho \mid \zeta' : \rho \in \Gamma \;\&\; \zeta' \neq \zeta\} \cup \{\zeta : \psi(\Gamma(\zeta))\} & \text{dependent row overriding} \\
\varphi \circ \varphi'(\Gamma) & = & \varphi'(\varphi(\Gamma)) & \text{composition}
\end{array}
$$

Fig. 10. Effects

ι, which are expressions), a set of *logical addresses* (LogAddress). The set LogAddress is a denumerable set and it contains the distinguished element ι_{self}. The element ι_{self} represents the logical address of the current object (self). The remaining elements of LogAddress represent the logical addresses of clones: ι_{c} ranges over these elements. We use ζ to denote an element of PhysAddress $\uplus$LogAddress:

$$\zeta \quad ::= \quad \iota \mid \iota_{\mathsf{self}} \mid \iota_{\mathsf{c}}$$

Figure 8 lists the definitions of types, environments, and effects.

An *object type* is simply either an address (the physical or logical address of an object; the notation $\mathsf{Obj}(\zeta)$ is used to stress that ζ is looked at as a type of an object) or a "union" of addresses. The union operator is needed in order to type programs containing conditional expressions. Our type judgments will enable to infer, for any expression, an object type and an effect. In conditional expressions we need to record some way the fact that only one of the branches will be actually evaluated. Union types enable us to keep track of these two possibilities instead of having to restrict the system by imposing both branches be typable with the same object type. For what concerns effects, instead, the same effect will be required to be inferred for both branches of a conditional expression. This condition restricts the range of typable programs, but prevents a combinatorial

explosion of computational behaviours a type system could not cope with in a feasible way. Hence, the union operator on object types provides a good degree of expressibility in our type system, while the condition on effects of branches of conditional expressions maintain the system quite feasible. We shall consider object types modulo idempotency, commutativity and associativity of union. We define:

$$\mathcal{O}(\tau) = \{\zeta \mid \tau \equiv \mathsf{Obj}(\zeta) \vee \tau' \text{ for some } \tau'\}.$$

We have two kinds of *method type*. Methods added with eager update are fields containing objects: so they are assigned object types.
Methods added with lazy update are methods whose bodies are unevaluated expressions: we type them with triples of environments, object types and effects. These triples give complete type information about the typing of the bodies: if b has type $\langle\Theta, \tau, \varphi\rangle$ then the judgment $\Theta \vdash \mathsf{b} : \tau, \varphi$ can be derived.
Method rows denote partial mappings between method names and method types.
Delegate rows denote partial mappings between delegate names and delegate types: notice that types in delegate rows must be singleton types, they cannot be unions of singleton types.
The order in which methods (respectively delegates) occur in the corresponding rows is irrelevant.
Row types are pairs of method rows and delegate rows: they show the methods and delegates of objects with their types.
Address Environments represent partial mappings between addresses and row types. They can be essentially seen as predicates (constraints) on stores. We shall use $\zeta \in \Gamma$ as short for $\exists\rho.\zeta : \rho \in \Gamma$.
Term Environments, instead, state which object type and effect a term variable is assumed to have in a given address environment. Term variables are introduced in order to type programs containing conditional expressions, which in a typing derivation are substituted for variables. We shall use $\Delta; x : \Gamma, \tau, \varphi$ as short for $\Delta \cup \{x : \Gamma, \tau, \varphi\}$.
Environments are pairs of address environments and term environments.
A *row operator* is a total function from row types to row types as defined in Figure 9. I.e. a row operator is one of the four operations: method deletion, delegate deletion, method update, and delegate update.
Effects denote total functions from address environments to address environments: they update the row types of addresses. This updating can be either an overriding or an addition, according to the presence of the address in the environment. The overriding can be dependent or independent from the actual row type of the current address. Dependent overriding uses the row operators. The effects are built by composition out of the three basic functions on environments: identity, independent row updating, dependent row overriding (see Figure 10). The independent row updating behaves differently when $\zeta = \iota_{\mathsf{self}}$ or $\zeta \neq \iota_{\mathsf{self}}$. The updating $\{[\iota_{\mathsf{self}} : \rho]\}$ adds the pair $\iota_{\mathsf{self}} : \rho$ to the address environment, possibly deleting a pair with first component ι_{self}. Instead updating $\{[\zeta : \rho]\}$ with $\zeta \neq \iota_{\mathsf{self}}$ adds the pair $\zeta : \rho$ to the address environment Γ only if $\zeta \notin \Gamma$. The dependent row overriding $\{\{\zeta : \psi\}\}$ must be applied to an address environment containing a pair $\zeta : \rho$: the resulting address environment will contain the pair $\zeta : \psi(\rho)$.

As we already said, a *typing judgment* has the shape:

$$\Gamma \mid \Delta \vdash \mathsf{a} : \tau, \varphi$$

where $\Gamma \mid \Delta$ is an environment, τ is an object type and φ is an effect.

B. Typing Rules

Figure 7 lists the typing rules. We use the following notational convention:

$$\langle\!\langle \mathsf{n} : \lambda \rangle\!\rangle \in \rho \iff \rho \equiv \langle\!\langle \mathsf{n} : \lambda \rangle\!\rangle \mid \nu' \parallel \mu \text{ or } \rho \equiv \nu \parallel \langle\!\langle \mathsf{n} : \lambda \rangle\!\rangle \mid \mu'$$

Axiom (Ax-x-init) allows to use the assumptions in the term environments.
The axioms (Ax-ι-init), (Ax-ι) and (Ax-self) give to a physical address and to self the types representing them. The axiom (Ax-ι-init) adds ι to the address environment with the empty row. The other two axioms do not change the address environment, so their effect is id.

Rule (R-clone) requires a singleton type for the expression a. This rule says that $\mathsf{clone}(\mathsf{a})$ has a fresh logical address and the row type of a. Notice that the row type of a is taken from the address environment obtained from the initial one by applying the effect of the typing of a.

The composition rule (R-comp) types the second expression in the address environment obtained from the initial one by applying the effect of the typing of the first expression.

The method selection rules (R-eager-sel) and (R-lazy-sel) requires a singleton type for the receiver. These rules look for the method in the whole up hierarchy of the delegates of the object: the typing is successful only when we find at most one occurrence of the method in the search procedure for it (i.e. the function $\mathcal{L}$ of Figure 11 produces a singleton). Such a look up procedure is described by means of the function $\mathcal{L}t$ which uses the function $\mathcal{L}$: both functions are defined in Figure 11. The current address environment is obtained from the initial one taking into account the changes due to the typing of the receiver.

If the resulting type is an object type τ this means that the method was updated in an eager way, i.e. the body of the method has been evaluated before the method updating. Rule (R-eager-sel) gives to the expression $\mathsf{a.m}$ the address ζ of the receiver a in case τ is the logical address ι_{self} of self. We denote this type by $\tau[\zeta/\iota_{\mathsf{self}}]$. The final effect is the effect of the typing of a.

Instead if the type of the method body is a triple $\langle \Gamma' \mid \Delta', \tau, \varphi' \rangle$ then the method was updated in a lazy way, i.e. the body b of the method has not been evaluated before the method updating. We must then check that b is well typed in the current environment, i.e. in the address environment $\varphi(\Gamma)$ obtained from the initial one (Γ) by applying the effect φ of the typing of a, and in the term environment Δ. We know b is well typed in the environment $\Gamma' \mid \Delta'$, so we have to check that $\varphi(\Gamma) \mid \Delta$ is as good as $\Gamma' \mid \Delta'$. To compare address environments we introduce a partial order on them. We say that Γ is better than Γ' (notation $\Gamma \leq \Gamma'$) iff the binary function $\mathcal{L}t(\Gamma, \_, \_)$ is an extension of $\mathcal{L}t(\Gamma', \_, \_)$, that is:

$$\Gamma \leq \Gamma' \quad \text{iff}$$
$$\mathcal{L}t(\Gamma', \zeta, \mathsf{n}) \neq \mathcal{U}df \implies \mathcal{L}t(\Gamma', \zeta, \mathsf{n}) = \mathcal{L}t(\Gamma, \zeta, \mathsf{n})$$

We have also to take into account that a is the receiver of the method, so we modify $\varphi(\Gamma)$ by associating to the logical address ι_{self} of self the row type ρ of the receiver a (i.e. we consider the environment $\varphi \circ \{[\iota_{\mathsf{self}} : \rho]\}(\Gamma)$). To sum up we have to compare $\varphi \circ \{[\iota_{\mathsf{self}} : \rho]\}(\Gamma)$ with Γ': this gives the condition $\varphi \circ \{[\iota_{\mathsf{self}} : \rho]\}(\Gamma) \leq \Gamma'$. For term environments we simply require Δ' to be a subset of Δ, i.e. $\Delta' \subseteq \Delta$. The remaining condition $\{\iota_{\mathsf{c}} \mid \iota_{\mathsf{c}} \in \varphi'(\Gamma') \ \& \ \iota_{\mathsf{c}} \in \varphi(\Gamma) \ \& \ \iota_{\mathsf{c}} \notin \Gamma'\} = \emptyset$ assures that there are no name clashes between the logical addresses of clones which are created independently either typing a or typing b. The resulting type is $\tau[\zeta/\iota_{\mathsf{self}}]$ as in rule (R-eager-sel). The resulting effect $\varphi \circ \varphi'[\zeta/\iota_{\mathsf{self}}]$ is the composition of the effect φ of typing a with the effect φ' of typing b where the address of self has been replaced by the address of a.

The rules for delegate selection (R-del-eager-sel) and (R-del-lazy-sel) differ from the corresponding selection rules only in the way of looking for the method type. These rules first look for the delegate d in the upper hierarchy of the delegates of the object. If the delegate d is found the rules look for the method in the upper hierarchy of the delegates of d.

To type an eager or lazy method updating we have to look for the address of a delegate containing the method in the whole upper hierarchy of the delegates of the object. This is done in rules (R-eager-up) and (R-lazy-up) by means of the function $\mathcal{L}a$ which uses the function $\mathcal{L}$: both functions are defined in Figure 11. Notice that if no delegate contains the required method the function $\mathcal{L}a$ returns the address of the receiver and the updating extends the receiver: otherwise the updating is an overriding.

In rule (R-eager-up) the new method body is typed in the address environment obtained from the initial one by applying the effect of the typing of the receiver and by associating to the address of self the row of the object at the address ζ' obtained by $\mathcal{L}a$. The resulting type is the type of the receiver and the resulting effect is the composition of the effect φ of typing the receiver with the effect φ' of typing the method body and with the updating of the row ρ of the address ζ'. This updating amounts either to override the type of method m in ρ by the type τ of the new body or to extend ρ with $\mathsf{m} : \tau$. Using the notations of Figures 9 and 10 the effect representing this updating is $\{\{\zeta' : \langle\!\langle \mathsf{m} : \tau \rangle\!\rangle_{\bullet}\}\}$.

Rule (R-lazy-up) instead only requires that the new method body b be typable in some environment Θ, which may have no relation with the environment $\Gamma \mid \Delta$ in which the receiver is typed. This is safe, since the type of the added method will recall Θ and the typing rule for method call (R-lazy-sel) will check that the environment in which the method call is typed will be better than Θ. If we can derive $\Theta \vdash \mathsf{b} : \tau, \varphi'$ then the method m will get the type $\langle \Theta, \tau, \varphi' \rangle$; this type gives a complete information about the typing of b. The type of $\mathsf{a.m} \lhd \mathsf{b}$ is the type of the receiver. The resulting effect is the composition of the effect φ of typing the receiver with the updating of the row ρ of the address ζ' (value of $\mathcal{L}a(\varphi(\Gamma), \zeta, \mathsf{m})$). This updating amounts either to override the type of method m in ρ by the type $\langle \Gamma', \tau, \varphi' \rangle$ or to extend ρ with $\mathsf{m} : \langle \Gamma', \tau, \varphi' \rangle$. Using the notations of Figures 9 and 10 the effect representing this updating is $\{\{\zeta' : \langle\!\langle \mathsf{m} : \langle \Gamma', \tau, \varphi' \rangle \rangle\!\rangle_{\bullet}\}\}$.

For typing a delegate updating rule (R-del-up) requires to deduce a singleton type for both the receiver and the new delegate: the delegate must be typed in the address environment obtained from the initial one by applying the effect of the typing of the receiver. The resulting type is the singleton type of the receiver and the resulting effect is the composition of the effect φ of typing the receiver with the effect φ' of typing the body and with the updating of the row ρ of the receiver. This updating amounts either to override the type of delegate d in ρ by the type $\mathsf{Obj}(\zeta')$ of the new delegate or to extend ρ with $\mathsf{d} : \mathsf{Obj}(\zeta')$. Using the notations of Figures 9 and 10 the effect representing this updating is $\{\{\zeta : \langle\!\langle \mathsf{d} : \mathsf{Obj}(\zeta') \rangle\!\rangle_{@}\}\}$ where ζ is the address of the receiver.

Rules (R-met-rem) and (R-del-rem) type respectively method remove and delegate remove. Again the receiver must be typed with a singleton type which is resulting type. The resulting effect is the composition of the effect φ of typing the receiver with the deletion of the method (in rule (R-met-rem)) or with the deletion of

the delegate (in rule (R-del-rem)) in the row of the receiver. Using the notations of Figure 9 the effects representing these deletions are respectively $\{\{\zeta : /_\bullet \mathsf{m}\}\}$ and $\{\{\zeta : /_@ \mathsf{d}\}\}$, where ζ is the address of the receiver.

Rules (R-union-I) and (R-union-E) deal with union types and are needed in order to type terms containing conditional expressions. Their shape closely resemble that of the logical rules for the disjunction in natural deduction systems.

Rule (R-union-I) requires that the evaluation of the condition does not change the address environment, i.e. the effect of the condition must be the identity. This rule, by introducing a union type in the conclusion, takes into account the types of both branches in a conditional expression. Notice that, as discussed before, we leave the possibility for a and a' to have different object types. However, in order to prevent an unmanageable combinatorial explosion, our type system imposes the effects derived for both a and a' to be the same.

Rule (R-union-E) is used in order to actually type a term containing a subexpression, which must be a conditional expression, having a union type. The requirement of replacing term variables only by conditional expressions does not reduce the set of typable terms in a significant way, since we derive union types only for conditional expressions. We chose the actual formulation to simplify the proof of Theorem 1.

Rules (R-cond-t) and (R-cond-f) can type conditional expressions when the type of the condition already provides the information needed to choose a branch. We require typability also for the discharged branch, in order to get that all sub-expressions of a well typed expression will be well typed too. Also these rules like (R-union-I) require that the evaluation of the condition does not change the address environment, but the effects derived for a and a' can be different.

C. Typing Examples

In this subsection we will use $\Gamma \vdash \ldots$ as a shorthand for $\Gamma \mid \emptyset \vdash \ldots$.

We first give a detailed outline of the typing for the δ^+ expression:

$$\iota@\mathsf{d}_1 \blacktriangleleft \iota'; \iota@\mathsf{d}_2 \blacktriangleleft \iota''; \iota'.\mathsf{m} \lhd (\iota''.\mathsf{m}' \blacktriangleleft \mathsf{self}); \iota.\mathsf{m}; \iota.\mathsf{m}'$$

considered in section II.

The overall typing is a series of applications of the rule (R-comp) (see Figure 12 where $\tau \equiv \mathsf{Obj}(\iota'')$). The typing of $\emptyset \vdash \iota@\mathsf{d}_1 \blacktriangleleft \iota' : \mathsf{Obj}(\iota), \varphi_1$ uses rule (R-del-up) producing effect:

$$\varphi_1 = \{[\iota : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle]\} \circ \{[\iota' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle]\} \circ \{\{\iota : \langle\!\langle \mathsf{d}_1 : \mathsf{Obj}(\iota')\rangle\!\rangle_@\}\}$$

which states that ι and ι' are empty object types, (produced by rule (Ax-ι-init)), and ι' is the delegate d_1 of ι. Rule (R-comp) requires the effect φ_1 to be applied to the empty environment to produce address environment:

$$\begin{array}{rcl} \Gamma_1 & = & \{\iota : \langle\!\langle \mathsf{d}_1 : \mathsf{Obj}(\iota')\rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle, \\ & & \iota' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle\} \end{array}$$

Similarly in typing $\Gamma_1 \vdash \iota@\mathsf{d}_2 \blacktriangleleft \iota'' : \mathsf{Obj}(\iota), \varphi_2$ we use (R-del-up) and obtain effect:

$$\varphi_2 = \{[\iota'' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle]\} \circ \{\{\iota : \langle\!\langle \mathsf{d}_2 : \mathsf{Obj}(\iota'')\rangle\!\rangle_@\}\}$$

Applying φ_2 to address environment Γ_1 we get:

$$\begin{array}{rcl} \Gamma_2 & = & \{\iota : \langle\!\langle \mathsf{d}_1 : \mathsf{Obj}(\iota') \mid \mathsf{d}_2 : \mathsf{Obj}(\iota'')\rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle, \\ & & \iota' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle, \\ & & \iota'' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle\} \end{array}$$

The typing of $\Gamma_2 \vdash \iota'.\mathsf{m} \lhd (\iota''.\mathsf{m}' \blacktriangleleft \mathsf{self}) : \mathsf{Obj}(\iota'), \varphi_3$ (see Figure 13) uses rules (R-eager-up), (R-lazy-up) and we obtain effect:

$$\varphi_3 = \{\{\iota' : \langle\!\langle \mathsf{m} : \langle \emptyset \mid \emptyset, \mathsf{Obj}(\iota''), \varphi_5\rangle\rangle\!\rangle_\bullet\}\}$$

where

$$\varphi_5 = \varphi_6 \circ \{\{\iota'' : \langle\!\langle \mathsf{m}' : \mathsf{Obj}(\iota_{\mathsf{self}})\rangle\!\rangle_\bullet\}\}$$

and

$$\varphi_6 = \{[\iota'' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle]\}.$$

φ_3 states that method m is updated at ι' with a body of type $\langle \emptyset \mid \emptyset, \mathsf{Obj}(\iota''), \varphi_5\rangle$. Applying φ_3 to address environment Γ_2 we get:

$$\begin{array}{rcl} \Gamma_3 & = & \{\iota : \langle\!\langle \mathsf{d}_1 : \mathsf{Obj}(\iota') \mid \mathsf{d}_2 : \mathsf{Obj}(\iota'')\rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle, \\ & & \iota' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle \mathsf{m} : \langle \emptyset \mid \emptyset, \mathsf{Obj}(\iota''), \varphi_5\rangle\rangle\!\rangle, \\ & & \iota'' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle\} \end{array}$$

Address environment Γ_3 now reflects the fact that ι' has method m with the type given above.

The typing of $\Gamma_3 \vdash \iota.\mathsf{m} : \mathsf{Obj}(\iota''), \varphi_4$ uses rule (R-lazy-sel) given that m was lazily inserted into ι'. The resulting effect is:

$$\varphi_4 = \varphi_5[\iota/\iota_{\mathsf{self}}] = \varphi_6 \circ \{\{\iota'' : \langle\!\langle \mathsf{m}' : \mathsf{Obj}(\iota)\rangle\!\rangle_\bullet\}\}$$

$$\begin{array}{rcl} \mathcal{L}(\Gamma, \zeta, \mathsf{n}) & = & \begin{cases} \{(\lambda, \zeta)\} & \text{if } \langle\!\langle \mathsf{n} : \lambda\rangle\!\rangle \in \Gamma(\zeta) \\ \bigcup_{\zeta' \in \mathcal{I}} \mathcal{L}(\Gamma, \zeta', \mathsf{n}) & \text{otherwise} \end{cases} \\ & & \text{where } \mathcal{I} = \{\zeta' \mid \langle\!\langle \mathsf{d} : \mathsf{Obj}(\zeta')\rangle\!\rangle \in \Gamma(\zeta)\} \\ \mathcal{L}a(\Gamma, \zeta, \mathsf{n}) & = & \begin{cases} \zeta' & \text{if } \mathcal{L}(\Gamma, \zeta, \mathsf{n}) = \{(\lambda, \zeta')\} \\ \zeta & \text{otherwise} \end{cases} \\ \mathcal{L}t(\Gamma, \zeta, \mathsf{n}) & = & \begin{cases} \lambda & \text{if } \mathcal{L}(\Gamma, \zeta, \mathsf{n}) = \{(\lambda, \zeta')\} \\ \mathcal{U}df & \text{otherwise} \end{cases} \end{array}$$

Fig. 11. The $\mathcal{L}$, $\mathcal{L}a$ and $\mathcal{L}t$ Functions

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \iota@\mathsf{d}_1 \blacktriangleleft \iota' : \mathsf{Obj}(\iota), \varphi_1} \quad \dfrac{}{\Gamma_1 \vdash \iota@\mathsf{d}_2 \blacktriangleleft \iota'' : \mathsf{Obj}(\iota), \varphi_2} \quad \dfrac{}{\Gamma_2 \vdash \iota'.\mathsf{m} \lhd (\iota''.\mathsf{m}' \blacktriangleleft \mathsf{self}) : \mathsf{Obj}(\iota'), \varphi_3} \quad \dfrac{}{\Gamma_3 \vdash \iota.\mathsf{m} : \tau, \varphi_4}}{\emptyset \vdash \iota@\mathsf{d}_1 \blacktriangleleft \iota'; \iota@\mathsf{d}_2 \blacktriangleleft \iota''; \iota'.\mathsf{m} \lhd (\iota''.\mathsf{m}' \blacktriangleleft \mathsf{self}); \iota.\mathsf{m} : \tau, \varphi_2 \circ \varphi_3 \circ \varphi_4} \quad \dfrac{}{\Gamma_4 \vdash \iota.\mathsf{m}' : \tau, \mathsf{id}}}{\emptyset \vdash \iota@\mathsf{d}_1 \blacktriangleleft \iota'; \iota@\mathsf{d}_2 \blacktriangleleft \iota''; \iota'.\mathsf{m} \lhd (\iota''.\mathsf{m}' \blacktriangleleft \mathsf{self}); \iota.\mathsf{m}; \iota.\mathsf{m}' : \tau, \varphi_1 \circ \varphi_2 \circ \varphi_3 \circ \varphi_4}$$

Fig. 12. Typing of $\iota@\mathsf{d}_1 \blacktriangleleft \iota'; \iota@\mathsf{d}_2 \blacktriangleleft \iota''; \iota'.\mathsf{m} \lhd (\iota''.\mathsf{m}' \blacktriangleleft \mathsf{self}); \iota.\mathsf{m}; \iota.\mathsf{m}'$

$$\dfrac{\dfrac{}{\Gamma_2 \vdash \iota' : \mathsf{Obj}(\iota'), \mathsf{id}} \quad \dfrac{\dfrac{}{\emptyset \vdash \iota'' : \mathsf{Obj}(\iota''), \varphi_6} \quad \varphi_6 \circ \{[\iota_{\mathsf{self}} : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle]\}(\emptyset) \vdash \mathsf{self} : \mathsf{Obj}(\iota_{\mathsf{self}}), \mathsf{id}}{\emptyset \vdash \iota''.\mathsf{m}' \blacktriangleleft \mathsf{self} : \mathsf{Obj}(\iota''), \varphi_5}\ \text{(R-eager-up)}}{\Gamma_2 \vdash \iota'.\mathsf{m} \lhd (\iota''.\mathsf{m}' \blacktriangleleft \mathsf{self}) : \mathsf{Obj}(\iota'), \varphi_3}\ \text{(R-lazy-up)}$$

Fig. 13. Typing of $\iota'.\mathsf{m} \lhd (\iota''.\mathsf{m}' \blacktriangleleft \mathsf{self})$

Applying φ_4 to address environment Γ_3 we get:

$$\begin{array}{rcl}\Gamma_4 & = & \{\iota : \langle\!\langle \mathsf{d}_1 : \mathsf{Obj}(\iota') \mid \mathsf{d}_2 : \mathsf{Obj}(\iota'')\rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle, \\ & & \iota' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle \mathsf{m} : \langle \emptyset \mid \emptyset, \mathsf{Obj}(\iota''), \varphi_5\rangle\rangle\!\rangle, \\ & & \iota'' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle \mathsf{m}' : \mathsf{Obj}(\iota)\rangle\!\rangle\}\end{array}$$

Address environment Γ_4 now reflects the fact that ι'' has method m' with type $\mathsf{Obj}(\iota)$.
The typing of $\Gamma_4 \vdash \iota.\mathsf{m}' : \mathsf{Obj}(\iota), \mathsf{id}$ uses rule (R-eager-sel) given that m' was eagerly inserted into ι'' as described in effect φ_4.

We now give two further examples demonstrating the typing of expressions containing conditionals. The first example expression demonstrates using rule (R-cond-f):

$$\iota.\mathsf{m} \lhd (\mathsf{self}); (\mathsf{if}\ \iota_{\mathsf{f}}\ \mathsf{then}\ (\iota'@\mathsf{d}_1 \blacktriangleleft \iota)\ \mathsf{else}\ (\iota''@\mathsf{d}_2 \blacktriangleleft \iota)).\mathsf{m}$$

We type the expression in environment Γ_0 containing ι_{f} so that when typing the conditional test we have effect id. Environment Γ_0 is defined as:

$$\Gamma_0 = \{\iota_{\mathsf{f}} : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle\}$$

The overall typing is shown in Figure 14.
The typing of $\emptyset \vdash \iota.\mathsf{m} \lhd (\mathsf{self}) : \mathsf{Obj}(\iota), \varphi_1$ uses rule (R-lazy-up) as to give effect:

$$\varphi_1 = \{[\iota : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle]\} \circ \{\{\iota : \langle\!\langle \mathsf{m} : \langle \Gamma_s \mid \emptyset, \mathsf{Obj}(\iota_{\mathsf{self}}), \mathsf{id}\rangle\rangle\!\rangle_\bullet\}\}$$

We are required by (R-lazy-up) to store an environment that can type self, we can use Γ_s:

$$\Gamma_s = \{\iota_{\mathsf{self}} : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle\}$$

Applying φ_1 to Γ_0 gives address environment:

$$\begin{array}{rcl}\Gamma_1 & = & \{\iota_{\mathsf{f}} : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle, \\ & & \iota : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle \mathsf{m} : \langle \Gamma_s \mid \emptyset, \mathsf{Obj}(\iota_{\mathsf{self}}), \mathsf{id}\rangle\rangle\!\rangle\}\end{array}$$

In typing method call:

$$(\mathsf{if}\ \iota_{\mathsf{f}}\ \mathsf{then}\ (\iota'@\mathsf{d}_1 \blacktriangleleft \iota)\ \mathsf{else}\ (\iota''@\mathsf{d}_2 \blacktriangleleft \iota)).\mathsf{m}$$

we first type the receiver using rule (R-cond-f) as the conditional test, ι_{f}, has type $\mathsf{Obj}(\iota_{\mathsf{f}})$. In typing $\iota''@\mathsf{d}_2 \blacktriangleleft \iota$ we obtain type $\mathsf{Obj}(\iota'')$ and effect:

$$\varphi_2 = \{[\iota'' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle]\} \circ \{\{\iota'' : \langle\!\langle \mathsf{d}_2 : \mathsf{Obj}(\iota)\rangle\!\rangle_@\}\}$$

The derivation for the true branch is similar with an effect reflecting the addition of delegate d_1: we represent it by "$\cdots$" in Figure 14. Applying φ_2 to the address environment Γ_1 we get:

$$\begin{array}{rcl}\Gamma_2 & = & \{\iota_{\mathsf{f}} : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle, \\ & & \iota : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle \mathsf{m} : \langle \Gamma_s \mid \emptyset, \mathsf{Obj}(\iota), \mathsf{id}\rangle\rangle\!\rangle, \\ & & \iota'' : \langle\!\langle \mathsf{d}_2 : \mathsf{Obj}(\iota)\rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle\}\end{array}$$

Returning to the method call we lookup m in ι'' given address environment Γ_2. Given that ι'' has no method m it will be delegated to ι to get a body with type: $\langle \Gamma_s \mid \emptyset, \mathsf{Obj}(\iota_{\mathsf{self}}), \mathsf{id}\rangle$. Therefore, we are dealing with a lazy method call of type $\mathsf{Obj}(\iota'')$.

Figure 15 shows the typing of another expression containing a conditional:

$$\begin{array}{c}\iota.\mathsf{m}_1 \lhd (\mathsf{if}\ \mathsf{self}\ \mathsf{then}\ \mathsf{self}\ \mathsf{else}\ \iota'); \\ \iota.\mathsf{m}_2 \blacktriangleleft \iota''; \iota'@\mathsf{d}_1 \blacktriangleleft \iota; \iota.\mathsf{m}_1.\mathsf{m}_2\end{array}$$

$$\dfrac{\Gamma_0 \vdash \iota.\mathsf{m} \lhd (\mathsf{self}) : \mathsf{Obj}(\iota), \varphi_1 \quad \dfrac{\dfrac{\Gamma_1 \vdash \iota_\mathsf{f} : \mathsf{Obj}(\iota_\mathsf{f}), \mathsf{id} \quad \cdots \quad \Gamma_1 \vdash \iota''@\mathsf{d}_2 \blacktriangleleft \iota : \mathsf{Obj}(\iota''), \varphi_2}{\Gamma_1 \vdash (\mathsf{if}\ (\iota_\mathsf{f})\ \mathsf{then}\ (\iota'@\mathsf{d}_1 \blacktriangleleft \iota)\ \mathsf{else}\ (\iota''@\mathsf{d}_2 \blacktriangleleft \iota)) : \mathsf{Obj}(\iota''), \varphi_2}\,\text{(R-cond-f)}}{\Gamma_1 \vdash (\mathsf{if}\ (\iota_\mathsf{f})\ \mathsf{then}\ (\iota'@\mathsf{d}_1 \blacktriangleleft \iota)\ \mathsf{else}\ (\iota''@\mathsf{d}_2 \blacktriangleleft \iota)).\mathsf{m} : \mathsf{Obj}(\iota''), \varphi_2}\,\text{(R-lazy-sel)}}{\Gamma_0 \vdash \iota.\mathsf{m} \lhd (\mathsf{self}); (\mathsf{if}\ (\iota_\mathsf{f})\ \mathsf{then}\ (\iota'@\mathsf{d}_1 \blacktriangleleft \iota)\ \mathsf{else}\ (\iota''@\mathsf{d}_2 \blacktriangleleft \iota)).\mathsf{m} : \mathsf{Obj}(\iota''), \varphi_1 \circ \varphi_2}\,\text{(R-comp)}$$

Fig. 14. Typing of $\iota.\mathsf{m} \lhd (\mathsf{self}); (\mathsf{if}\ \iota_\mathsf{f}\ \mathsf{then}\ (\iota'@\mathsf{d}_1 \blacktriangleleft \iota)\ \mathsf{else}\ (\iota''@\mathsf{d}_2 \blacktriangleleft \iota)).\mathsf{m}$

$$\dfrac{\begin{array}{l}\emptyset \mid \Delta_1 \vdash \iota.\mathsf{m}_1 \lhd x; \iota.\mathsf{m}_2 \blacktriangleleft \iota''; \iota'@\mathsf{d}_1 \blacktriangleleft \iota; \iota.\mathsf{m}_1.\mathsf{m}_2 : \mathsf{Obj}(\iota''), \varphi_4 \\ \emptyset \mid \Delta_2 \vdash \iota.\mathsf{m}_1 \lhd x; \iota.\mathsf{m}_2 \blacktriangleleft \iota''; \iota'@\mathsf{d}_1 \blacktriangleleft \iota; \iota.\mathsf{m}_1.\mathsf{m}_2 : \mathsf{Obj}(\iota''), \varphi_4 \\ \Gamma_1 \vdash \mathsf{self} : \mathsf{Obj}(\iota_\mathsf{self}), \mathsf{id} \\ \Gamma_1 \vdash \mathsf{self} : \mathsf{Obj}(\iota_\mathsf{self}), \mathsf{id} \qquad \Gamma_1 \vdash \iota' : \mathsf{Obj}(\iota'), \mathsf{id}\end{array}}{\emptyset \vdash \iota.\mathsf{m}_1 \lhd (\mathsf{if\ self\ then\ self\ else}\ \iota'); \iota.\mathsf{m}_2 \blacktriangleleft \iota''; \iota'@\mathsf{d}_1 \blacktriangleleft \iota; \iota.\mathsf{m}_1.\mathsf{m}_2 : \mathsf{Obj}(\iota''), \varphi_4}\,\text{(R-union-E)}$$

Fig. 15. Typing of $\iota.\mathsf{m}_1 \lhd (\mathsf{if\ self\ then\ self\ else}\ \iota'); \iota.\mathsf{m}_2 \blacktriangleleft \iota''; \iota'@\mathsf{d}_1 \blacktriangleleft \iota; \iota.\mathsf{m}_1.\mathsf{m}_2$

$$\dfrac{\emptyset \mid \Delta_1 \vdash \iota.\mathsf{m}_1 \lhd x : \mathsf{Obj}(\iota), \varphi_1 \qquad \dfrac{\Gamma_2 \mid \Delta_1 \vdash \iota.\mathsf{m}_2 \blacktriangleleft \iota'' : \mathsf{Obj}(\iota), \varphi_2 \qquad \dfrac{\Gamma_3 \mid \Delta_1 \vdash \iota'@\mathsf{d}_1 \blacktriangleleft \iota : \mathsf{Obj}(\iota'), \varphi_3 \quad \Gamma_4 \mid \Delta_1 \vdash \iota.\mathsf{m}_1.\mathsf{m}_2 : \mathsf{Obj}(\iota''), \mathsf{id}}{\Gamma_3 \mid \Delta_1 \vdash \iota'@\mathsf{d}_1 \blacktriangleleft \iota; \iota.\mathsf{m}_1.\mathsf{m}_2 : \mathsf{Obj}(\iota''), \varphi_3}}{\Gamma_2 \mid \Delta_1 \vdash \iota.\mathsf{m}_2 \blacktriangleleft \iota''; \iota'@\mathsf{d}_1 \blacktriangleleft \iota; \iota.\mathsf{m}_1.\mathsf{m}_2 : \mathsf{Obj}(\iota''), \varphi_2 \circ \varphi_3}}{\emptyset \mid \Delta_1 \vdash \iota.\mathsf{m}_1 \lhd x; \iota.\mathsf{m}_2 \blacktriangleleft \iota''; \iota'@\mathsf{d}_1 \blacktriangleleft \iota; \iota.\mathsf{m}_1.\mathsf{m}_2 : \mathsf{Obj}(\iota''), \varphi_1 \circ \varphi_2 \circ \varphi_3}$$

Fig. 16. Typing of $\iota.\mathsf{m}_1 \lhd x; \iota.\mathsf{m}_2 \blacktriangleleft \iota''; \iota'@\mathsf{d}_1 \blacktriangleleft \iota; \iota.\mathsf{m}_1.\mathsf{m}_2$

However, this time we will use a union type for the conditional, (if self then self else ι'), i.e. type $\mathsf{Obj}(\iota_\mathsf{self}) \vee \mathsf{Obj}(\iota')$. This reflects the fact that the true branch has type $\mathsf{Obj}(\iota_\mathsf{self})$ and the false branch has type $\mathsf{Obj}(\iota')$. In order to type the whole expression we type it with the conditional expression replaced with term variable x:

$$\iota.\mathsf{m}_1 \lhd x; \iota.\mathsf{m}_2 \blacktriangleleft \iota''; \iota'@\mathsf{d}_1 \blacktriangleleft \iota; \iota.\mathsf{m}_1.\mathsf{m}_2$$

This will allow us to apply (R-union-E), which will require us to have two derivations one with term environment

$$\Delta_1 = \{x : \Gamma_1, \mathsf{Obj}(\iota_\mathsf{self}), \mathsf{id}\}$$

and the other with term environment

$$\Delta_2 = \{x : \Gamma_1, \mathsf{Obj}(\iota'), \mathsf{id}\}.$$

Address environment, Γ_1, is defined as:

$$\Gamma_1 \quad = \quad \{\iota_\mathsf{self} : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle, \\ \iota' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle\}$$

Figure 16 shows the typing of

$$\iota.\mathsf{m}_1 \lhd x; \iota.\mathsf{m}_2 \blacktriangleleft \iota''; \iota'@\mathsf{d}_1 \blacktriangleleft \iota; \iota.\mathsf{m}_1.\mathsf{m}_2$$

with term environment Δ_1. As in previous examples there is a series of applications of rule (R-comp). The typing of expression, $\iota.\mathsf{m}_1 \lhd x$, uses rule (R-lazy-up) to give effect:

$$\varphi_1 = \{[\![\iota : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle]\!] \circ \{\{\iota : \langle\!\langle \mathsf{m}_1 : \langle \Gamma_1 \mid \Delta_1, \mathsf{Obj}(\iota_\mathsf{self}), \mathsf{id} \rangle \rangle\!\rangle_\bullet\}\}$$

which states that method m_1 is updated at ι with a body of type $\langle \Gamma_1 \mid \Delta_1, \mathsf{Obj}(\iota_\mathsf{self}), \mathsf{id} \rangle$. We choose address environment Γ_1 as when typing x with rule (Ax-x-init)

we require that the current address environment is the same as that stored in Δ_1 for x. Applying φ_1 to the empty environment we get:

$$\Gamma_2 = \{\iota : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle \mathsf{m}_1 : \langle \Gamma_1 \mid \Delta_1, \mathsf{Obj}(\iota_{\mathsf{self}}), \mathsf{id}\rangle \rangle\!\rangle\}$$

When typing expression, $\iota.\mathsf{m}_2 \blacktriangleleft \iota''$ we use (R-eager-up) to give effect:

$$\varphi_2 = \{[\iota'' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle]\} \circ \{\{\iota : \langle\!\langle \mathsf{m}_2 : \mathsf{Obj}(\iota'') \rangle\!\rangle_\bullet\}\}$$

which states that method m_2 is updated at ι with a body of type $\mathsf{Obj}(\iota'')$ and ι'' is the empty object. Applying φ_2 to Γ_2 we get:

$$\begin{array}{rl} \Gamma_3 = & \{\iota : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle \mathsf{m}_1 : \langle \Gamma_1 \mid \Delta_1, \mathsf{Obj}(\iota_{\mathsf{self}}), \mathsf{id}\rangle \mid \mathsf{m}_2 : \mathsf{Obj}(\iota'') \rangle\!\rangle, \\ & \iota'' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle\} \end{array}$$

which reflects the fact that ι has methods m_1 and m_2 and ι'' is the empty object. Similarly, when typing expression $\iota'@\mathsf{d}_1 \blacktriangleleft \iota$ we obtain effect:

$$\varphi_3 = \{[\iota' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle]\} \circ \{\{\iota' : \langle\!\langle \mathsf{d}_1 : \mathsf{Obj}(\iota) \rangle\!\rangle_@\}\}$$

which states that delegate d_1 in ι' is updated with type $\mathsf{Obj}(\iota)$ and ι' is the empty object. Applying φ_3 to Γ_3 we get:

$$\begin{array}{rl} \Gamma_4 = & \{\iota : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle \mathsf{m}_1 : \langle \Gamma_1 \mid \Delta_1, \mathsf{Obj}(\iota_{\mathsf{self}}), \mathsf{id}\rangle \mid \mathsf{m}_2 : \mathsf{Obj}(\iota'') \rangle\!\rangle, \\ & \iota' : \langle\!\langle \mathsf{d}_1 : \mathsf{Obj}(\iota) \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle, \\ & \iota'' : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle\} \end{array}$$

which reflects as well as those things from Γ_3 the fact that ι is the delegate of ι'. Finally in typing $\iota.\mathsf{m}_1.\mathsf{m}_2$ we type the receiver, $\iota.\mathsf{m}_1$ which uses rule (R-lazy-sel) as method m_1 was lazily added to ι. Note that the body of m_1 has type $\langle \Gamma_1 \mid \Delta_1, \mathsf{Obj}(\iota_{\mathsf{self}}), \mathsf{id}\rangle$ and $\{[\iota_{\mathsf{self}} : \rho]\}(\Gamma_4) \leq \Gamma_1$, where ρ is the row type of ι in Γ_4, i.e.

$$\rho = \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle \mathsf{m}_1 : \langle \Gamma_1 \mid \Delta_1, \mathsf{Obj}(\iota_{\mathsf{self}}), \mathsf{id}\rangle \mid \mathsf{m}_2 : \mathsf{Obj}(\iota'') \rangle\!\rangle.$$

Therefore, the type of $\iota.\mathsf{m}_1$ will be $\mathsf{Obj}(\iota)$ (given we replace occurrences of ι_{self} with ι). We now lookup method m_2 in ι to get the return type $\mathsf{Obj}(\iota'')$. The typing of

$$\iota.\mathsf{m}_1 \lhd x; \iota.\mathsf{m}_2 \blacktriangleleft \iota''; \iota'@\mathsf{d}_1 \blacktriangleleft \iota; \iota.\mathsf{m}_1.\mathsf{m}_2$$

with term environment Δ_2 will be almost the same with the difference that the result of $\iota.\mathsf{m}_1$ will have type $\mathsf{Obj}(\iota')$ and thus method m_2 will be delegated from ι' to ι.

Given both derivations, and the typing of the conditional test, true and false branches in Γ_1 we can apply (R-union-E) to get type $\mathsf{Obj}(\iota'')$ and effect $\varphi_4 = \varphi_1 \circ \varphi_2 \circ \varphi_3$.

D. Soundness

There are clear correspondences between the syntax of the calculus (Figure 2) and that of types (Figure 8), the look up functions of the operational semantics (Figure 4) and those of the typing rules (Figure 11), the store update (Figure 5) and the row operations (Figure 9). So a soundness result for the given type assignment is expected.

An *address mapping* $\mathcal{A}$ is a partial mapping from addresses to physical addresses which is injective for all arguments with the exception of ι_{self} and such that $\mathcal{A}(\iota) = \iota$ for all memory addresses ι in the domain of $\mathcal{A}$. An address mapping $\mathcal{A}'$ is better than another address mapping $\mathcal{A}$ (notation $\mathcal{A}' \gtrsim \mathcal{A}$) iff $\mathcal{A}'$ is an extension of $\mathcal{A}$, i.e. $\mathcal{A}'(\zeta) = \mathcal{A}(\zeta)$ for all addresses ζ in the domain of $\mathcal{A}$.

The key notion in the following development is the *soundness of* δ^+ *expressions*. This soundness is relative to a triple: address environments, object types and effects. A δ^+ *expression* a *is* $\langle \Gamma, \tau, \varphi \rangle$*-sound* iff when evaluated in a store which "agrees" with Γ (by means of a suitable address mapping) either diverges or evaluates to an object which "agrees" with τ and the resulting store "agrees" with φ. So we need to define also *agreement between stores, address mappings, effects and address environments*: a pair of stores agrees with an effect iff the effect "says" how to update the first store for obtaining the second one. A store σ agrees with an address environment Γ via the address mapping $\mathcal{A}$ iff the objects in the store have delegates and fields as required by the address environment. Moreover the method bodies are sound with respect to their typings when their variables are replaced by sound δ^+ expressions.

Definition 1:

1) A *store* σ *agrees with an address mapping* $\mathcal{A}$ (notation $\sigma \propto \mathcal{A}$) iff $\sigma(\mathsf{self}) = \mathcal{A}(\iota_{\mathsf{self}})$.
2) The *pre-agreement of a pair of stores* (σ, σ') *with an effect* φ *via the address mapping* $\mathcal{A}$ (notation $(\sigma, \sigma') \widetilde{\propto}^{\mathcal{A}} \varphi$) is defined by induction on φ in Figure 17 (using the notion of point 4).
3) A *pair of stores* (σ, σ') *agrees with an an effect* φ *via the address mapping* $\mathcal{A}$ (notation $(\sigma, \sigma') \propto^{\mathcal{A}} \varphi$) iff $\sigma \propto \mathcal{A}$, $\sigma' \propto \mathcal{A}$ and $(\sigma, \sigma') \widetilde{\propto}^{\mathcal{A}} \varphi$.
4) A *store* σ *agrees with an address environment* Γ *via the address mapping* $\mathcal{A}$ (notation $\sigma \propto^{\mathcal{A}} \Gamma$) iff $\sigma \propto \mathcal{A}$ and they satisfy the conditions of Figure 18.
5) A δ^+ *expression* a *is* $\langle \Gamma, \tau, \varphi \rangle$*-sound* iff for all $\mathcal{A}, \sigma$ such that $\sigma \propto^{\mathcal{A}} \Gamma$
   - either a, σ diverges

$$
\begin{array}{lcl}
(\sigma,\sigma')\widetilde{\propto}^{\mathcal{A}}\mathsf{id} & \iff & \forall \iota.\sigma(\iota)=\sigma'(\iota)\\
(\sigma,\sigma')\widetilde{\propto}^{\mathcal{A}}\{[\zeta:\rho]\} & \iff & \forall \iota\neq\mathcal{A}(\zeta).\sigma(\iota)=\sigma'(\iota)\ \&\ \sigma'\propto^{\mathcal{A}}\{\zeta:\rho\}\\
(\sigma,\sigma')\widetilde{\propto}^{\mathcal{A}}\{\{\zeta:\psi\}\} & \iff & \forall \iota\neq\mathcal{A}(\zeta).\sigma(\iota)=\sigma'(\iota)\ \&\\
 & & \forall\rho.\sigma\propto^{\mathcal{A}}\{\zeta:\rho\}\implies\sigma'\propto^{\mathcal{A}}\{\zeta:\psi(\rho)\}\\
(\sigma,\sigma')\widetilde{\propto}^{\mathcal{A}}\varphi'\circ\varphi'' & \iff & \exists\sigma''.(\sigma,\sigma'')\widetilde{\propto}^{\mathcal{A}}\varphi'\ \&\ (\sigma'',\sigma')\widetilde{\propto}^{\mathcal{A}}\varphi''
\end{array}
$$

Fig. 17. Definition of $(\sigma,\sigma')\widetilde{\propto}^{\mathcal{A}}\varphi$

$$
\begin{array}{lcl}
\mathcal{L}t(\Gamma,\zeta,\mathsf{n})=\tau & \implies & \mathcal{L}ook(\sigma,\mathcal{A}(\zeta),\mathsf{n})=\mathcal{A}(\zeta')\ \&\ \zeta'\in\mathcal{O}(\tau)\\
\mathcal{L}t(\Gamma,\zeta,\mathsf{m})=\langle\Gamma'\mid\Delta,\tau,\varphi\rangle & \implies & \mathcal{L}ook(\sigma,\mathcal{A}(\zeta),\mathsf{m})=\mathsf{b}\ \&\\
\&\Delta=\{x_i:\Gamma_i,\tau_i,\varphi_i\mid i\in I\} & & \text{if } \mathsf{b}_i \text{ is } \langle\Gamma_i,\tau_i,\varphi_i\rangle\text{-sound for all } i\in I\\
 & & \text{then } \mathsf{b}[\mathsf{b}_i/x_i\mid i\in I] \text{ is } \langle\Gamma',\tau,\varphi\rangle\text{-sound.}
\end{array}
$$

Fig. 18. Conditions for $\sigma\propto^{\mathcal{A}}\Gamma$

- or $\mathsf{a},\sigma\rightsquigarrow_{\delta}\mathcal{A}'(\zeta),\sigma'$ for some ζ, $\mathcal{A}'$, σ' such that $\zeta\in\mathcal{O}(\tau)$, $\mathcal{A}'\gtrsim\mathcal{A}$, and $(\sigma,\sigma')\propto^{\mathcal{A}'}\varphi$.

To show soundness of our type system we use a bunch of properties relating the various agreements, the partial order on address environments and the replacement of self. These properties follow quite easily from the definitions.

Lemma 1: 1) If $\sigma\propto^{\mathcal{A}}\Gamma$ and $\Gamma\leq\Gamma'$ then $\sigma\propto^{\mathcal{A}}\Gamma'$.
2) If $(\sigma,\sigma')\propto^{\mathcal{A}}\varphi$ and $\mathcal{A}'\gtrsim\mathcal{A}$ then $(\sigma,\sigma')\propto^{\mathcal{A}'}\varphi$.
3) If $(\sigma,\sigma')\propto^{\mathcal{A}}\varphi$ and $\sigma\propto^{\mathcal{A}}\Gamma$, then $\sigma'\propto^{\mathcal{A}}\varphi(\Gamma)$.
4) If $\mathcal{A}'\gtrsim\mathcal{A}$ then $\mathcal{A}'[\iota_{\mathsf{self}}\mapsto\iota]\gtrsim\mathcal{A}[\iota_{\mathsf{self}}\mapsto\iota]$.
5) If $\sigma\propto^{\mathcal{A}}\Gamma$ and $\mathcal{A}'=\mathcal{A}[\iota_{\mathsf{self}}\mapsto\mathcal{A}(\zeta)]$ then $\sigma[\mathsf{self}\mapsto\mathcal{A}(\zeta)]\propto^{\mathcal{A}'}\{[\iota_{\mathsf{self}}:\Gamma(\zeta)]\}(\Gamma)$.
6) If $(\sigma,\sigma')\propto^{\mathcal{A}}\varphi$, $\mathcal{A}'=\mathcal{A}[\iota_{\mathsf{self}}\mapsto\iota]$, and $\sigma(\mathsf{self})=\mathcal{A}(\zeta)$ then $(\sigma[\mathsf{self}\mapsto\iota],\sigma'[\mathsf{self}\mapsto\iota])\propto^{\mathcal{A}}\varphi[\zeta/\iota_{\mathsf{self}}]$.

The following lemma relates $\mathcal{L}a$ and $\mathcal{L}ook$ functions with store update: the proof is straightforward.

Lemma 2: Let $\sigma\propto^{\mathcal{A}}\Gamma$ and $\mathcal{L}a(\Gamma,\zeta,\mathsf{m})=\zeta'$.
1) $\forall\iota\neq\mathcal{A}(\zeta').\sigma(\iota)=\sigma\{\mathcal{A}(\zeta).\mathsf{m}\lhd^{+}\mathsf{b}\}(\iota)$.
2) $\mathcal{L}ook(\sigma\{\mathcal{A}(\zeta).\mathsf{m}\lhd^{+}\mathsf{b}\},\mathcal{A}(\zeta'),\mathsf{m})=\mathsf{b}$.

To deal with δ^{+} expressions containing conditional it is useful to introduce an equivalence between δ^{+} expressions relative to a given state.

Definition 2: Two δ^{+} expressions a *and* b *are equivalent with respect to a state σ* (notation $\mathsf{a}\sim^{\sigma}\mathsf{b}$) iff either both a,σ and b,σ diverge, or a,σ and b,σ converge to the same address or to `stuckErr`.

When evaluating a δ^{+} expression a in a given state each occurrence of a sub-expression of a is either unevaluated or evaluated in an unique state.

Definition 3: Let $\mathsf{a},\sigma\rightsquigarrow_{\delta}\iota,\sigma'$ and b be an occurrence of a sub-expression of a. The *evaluation state of* b *in* a *relative to* σ (notation $\mathcal{S}(\mathsf{b},\mathsf{a},\sigma)$) is the state σ_{b} such that $\mathsf{b},\sigma_{\mathsf{b}}\rightsquigarrow_{\delta}\iota,\sigma'_{\mathsf{b}}$ occurs in the derivation of $\mathsf{a},\sigma\rightsquigarrow_{\delta}\iota,\sigma'$, if such a judgement exists, and undefined otherwise.

From the reduction rules for conditionals (see Figure 3) the following lemma follows easily.

Lemma 3: Let $\mathsf{d}\equiv\mathsf{c}[\mathsf{if\ b\ then\ a\ else\ a}'/x]$ be a δ^{+} expression such that d,σ converges, then:
- if $\mathcal{S}(\mathsf{b},\mathsf{a},\sigma)$ is undefined then $\mathsf{d}\sim^{\sigma}\mathsf{c}[\mathsf{b}'/x]$, where b' is an arbitrary δ^{+} expression;
- if $\mathcal{S}(\mathsf{b},\mathsf{a},\sigma)=\sigma_{\mathsf{b}}$ then
  - if $\mathsf{b},\sigma_{\mathsf{b}}\rightsquigarrow_{\delta}\iota,\sigma_{\mathsf{b}}$ and $\iota\neq\iota_{\mathsf{f}}$, then $\mathsf{d}\sim^{\sigma}\mathsf{c}[\mathsf{a}/x]$;
  - if $\mathsf{b},\sigma_{\mathsf{b}}\rightsquigarrow_{\delta}\iota_{\mathsf{f}},\sigma_{\mathsf{b}}$ then $\mathsf{d}\sim^{\sigma}\mathsf{c}[\mathsf{a}'/x]$.

The type system is sound in the sense that a converging well typed δ^{+} expression, when evaluated

in a suitable store, returns an address which agrees with the type of the expression, and never returns `stuckErr`.

Theorem 1 (Soundness): If

$$\Gamma \mid \{x_i : \Gamma_i, \tau_i, \varphi_i \mid i \in I\} \vdash \mathsf{a} : \tau, \varphi$$

and b_i is $\langle \Gamma_i, \tau_i, \varphi_i \rangle$-sound for all $i \in I$, then $\mathsf{a}[\mathsf{b}_i/x_i \mid i \in I]$ is $\langle \Gamma, \tau, \varphi \rangle$-sound.

Proof: The proof is by structural induction on a. Let

$$\Delta = \{x_i : \Gamma_i, \tau_i, \varphi_i \mid i \in I\}$$

and $\mathcal{A}, \sigma$ be such that $\sigma \propto^{\mathcal{A}} \Gamma$. If e is an arbitrary expression, $\overline{\mathsf{e}}$ is shorthand for $\mathsf{e}[\mathsf{b}_i/x_i \mid i \in I]$. We also show that all sub-expressions of a are evaluated in stores which "agree" with the address environments used to type them, i.e.:

(1) *If* b *is an occurrence of a sub-expression of* a, *the typing judgment* $\Gamma' \mid \Delta' \vdash \mathsf{b} : \tau', \varphi'$ *occurs in the derivation of* $\Gamma \mid \Delta \vdash \mathsf{a} : \tau, \varphi$, *and* $\overline{\mathsf{a}}, \sigma$ *converges, then either* $\mathcal{S}(\overline{\mathsf{b}}, \overline{\mathsf{a}}, \sigma)$ *is undefined or* $\mathcal{S}(\overline{\mathsf{b}}, \overline{\mathsf{a}}, \sigma) \propto^{\mathcal{A}} \Gamma'$.

Let us consider the case when a is the selection of a delegate method added or overrided with a lazy updating and the last applied rule in the typing of a is rule (R-del-lazy-sel):

$$\frac{\begin{array}{l}\Gamma \mid \Delta \vdash \mathsf{a}' : \mathsf{Obj}(\zeta), \varphi \\ \mathcal{L}t(\varphi(\Gamma), \zeta, \mathsf{d}) = \mathsf{Obj}(\zeta') \\ \mathcal{L}t(\varphi(\Gamma), \zeta', \mathsf{m}) = \langle \Gamma' \mid \Delta', \tau', \varphi' \rangle \\ \varphi(\Gamma)(\zeta) = \rho \quad \varphi \circ \{\![\iota_{\mathsf{self}} : \rho]\!\}(\Gamma) \leq \Gamma' \quad \Delta' \subseteq \Delta \\ \{\iota_{\mathsf{c}} \mid \iota_{\mathsf{c}} \in \varphi'(\Gamma') \;\&\; \iota_{\mathsf{c}} \in \varphi(\Gamma) \;\&\; \iota_{\mathsf{c}} \notin \Gamma'\} = \emptyset\end{array}}{\Gamma \vdash \mathsf{a}'@\mathsf{d}.\mathsf{m} : \tau'[\zeta/\iota_{\mathsf{self}}], \varphi \circ \varphi'[\zeta/\iota_{\mathsf{self}}]}$$

In this case $\mathsf{a} \equiv \mathsf{a}'@\mathsf{d}.\mathsf{m}$. By induction $\Gamma \mid \Delta \vdash \mathsf{a}' : \mathsf{Obj}(\zeta), \varphi$ implies that either $\overline{\mathsf{a}'}, \sigma$ diverges or $\overline{\mathsf{a}'}, \sigma \rightsquigarrow_{\delta} \mathcal{A}'(\zeta), \sigma'$ for some $\mathcal{A}'$, σ' such that $\mathcal{A}' \gtrsim \mathcal{A}$, and $(\sigma, \sigma') \propto^{\mathcal{A}'} \varphi$. If $\overline{\mathsf{a}'}, \sigma$ diverges then also $\overline{\mathsf{a}}, \sigma$ diverges.

Otherwise by Lemma 1(3) $(\sigma, \sigma') \propto^{\mathcal{A}} \varphi$ and $\sigma \propto^{\mathcal{A}} \Gamma$ imply $\sigma' \propto^{\mathcal{A}} \varphi(\Gamma)$. By definition from $\mathcal{L}t(\varphi(\Gamma), \zeta, \mathsf{d}) = \mathsf{Obj}(\zeta')$ we get $\mathcal{L}ook(\sigma', \mathcal{A}'(\zeta), \mathsf{d}) = \mathcal{A}'(\zeta')$. Again by definition from $\mathcal{L}t(\varphi(\Gamma), \zeta', \mathsf{m}) = \langle \Gamma' \mid \Delta', \tau', \varphi' \rangle$ we get $\mathcal{L}ook(\sigma, \mathcal{A}'(\zeta'), \mathsf{m}) = \mathsf{b}$. Being $\Delta' \subseteq \Delta$ we get $\Delta' = \{x_j : \tau_j, \varphi_j \mid j \in J\}$ for some $J \subseteq I$. Since all the free variables of b are subjects in Δ' we have $\overline{\mathsf{b}} \equiv \mathsf{b}[\mathsf{b}_j/x_j \mid j \in J]$. Then for all σ_b, $\mathcal{A}_b$, such that $\sigma_b \propto^{\mathcal{A}_b} \Gamma'$ either $\overline{\mathsf{b}}, \sigma_b$ diverges or $\overline{\mathsf{b}}, \sigma_b \rightsquigarrow_{\delta} \mathcal{A}'_b(\zeta''), \sigma'_b$ for some $\mathcal{A}'_b$, σ'_b such that $\mathcal{A}'_b \gtrsim \mathcal{A}_b$, and $(\sigma_b, \sigma'_b) \propto^{\mathcal{A}'_b} \varphi'$.

Let $\sigma''' = \sigma'[\mathsf{self} \mapsto \mathcal{A}'(\zeta)]$ and $\mathcal{A}'' = \mathcal{A}'[\iota_{\mathsf{self}} \mapsto \mathcal{A}'(\zeta)]$: by Lemma 1(5) $\sigma' \propto^{\mathcal{A}'} \varphi(\Gamma)$ implies $\sigma''' \propto^{\mathcal{A}''} \varphi \circ \{\![\iota_{\mathsf{self}} : \rho]\!\}(\Gamma)$. Being $\varphi \circ \{\![\iota_{\mathsf{self}} : \rho]\!\}(\Gamma) \leq \Gamma'$ we get $\sigma''' \propto^{\mathcal{A}''} \Gamma'$ by Lemma 1(1): this together with induction shows clause (1). Moreover either $\overline{\mathsf{b}}, \sigma'''$ diverges or $\overline{\mathsf{b}}, \sigma''' \rightsquigarrow_{\delta} \mathcal{A}'''(\zeta''), \sigma''$ for some $\mathcal{A}'''$, σ'' such that $\mathcal{A}''' \gtrsim \mathcal{A}''$, and $(\sigma''', \sigma'') \propto^{\mathcal{A}'''} \varphi'$. If $\overline{\mathsf{b}}, \sigma'''$ diverges then also $\overline{\mathsf{a}}, \sigma$ diverges. If $\overline{\mathsf{b}}, \sigma''' \rightsquigarrow_{\delta} \mathcal{A}'''(\zeta''), \sigma''$ then $\overline{\mathsf{a}}, \sigma \rightsquigarrow_{\delta} \mathcal{A}'''(\zeta''), \sigma''[\mathsf{self} \mapsto \sigma(\mathsf{self})]$.

Let $\mathcal{A}^* = \mathcal{A}'''[\iota_{\mathsf{self}} \mapsto \sigma(\mathsf{self})]$: then it suffices to show:

(2) $\mathcal{A}^* \gtrsim \mathcal{A}$;
(3) $\mathcal{A}^*(\zeta''[\zeta/\iota_{\mathsf{self}}]) = \mathcal{A}'''(\zeta'')$;
(4) $(\sigma, \sigma''[\mathsf{self} \mapsto \sigma(\mathsf{self})]) \propto^{\mathcal{A}^*} \varphi \circ \varphi'[\zeta/\iota_{\mathsf{self}}]$.

For (2) from $\mathcal{A}^* = \mathcal{A}'''[\iota_{\mathsf{self}} \mapsto \sigma(\mathsf{self})]$ and $\mathcal{A}''' \gtrsim \mathcal{A}''$ we get $\mathcal{A}^* \gtrsim \mathcal{A}''[\iota_{\mathsf{self}} \mapsto \sigma(\mathsf{self})]$ by Lemma 1(4). Being $\sigma \propto \mathcal{A}'$ it holds $\sigma(\mathsf{self}) = \mathcal{A}'(\iota_{\mathsf{self}})$. From above and $\mathcal{A}'' = \mathcal{A}'[\iota_{\mathsf{self}} \mapsto \mathcal{A}'(\zeta)]$ we have $\mathcal{A}''[\iota_{\mathsf{self}} \mapsto \sigma(\mathsf{self})] = \mathcal{A}'$: this together with $\mathcal{A}' \gtrsim \mathcal{A}$ allows us to conclude $\mathcal{A}^* \gtrsim \mathcal{A}$.

For (3) if $\zeta'' = \iota_{\mathsf{self}}$ then $\mathcal{A}^*(\zeta''[\zeta/\iota_{\mathsf{self}}]) = \mathcal{A}^*(\zeta) = \mathcal{A}(\zeta)$ and $\mathcal{A}'''(\zeta'') = \mathcal{A}'''(\iota_{\mathsf{self}}) = \mathcal{A}''(\iota_{\mathsf{self}}) = \mathcal{A}'(\zeta) = \mathcal{A}(\zeta)$, taking into account that $\mathcal{A}^* \gtrsim \mathcal{A}$, $\mathcal{A}''' \gtrsim \mathcal{A}''$, $\mathcal{A}'' = \mathcal{A}'[\iota_{\mathsf{self}} \mapsto \mathcal{A}'(\zeta)]$, and $\mathcal{A}' \gtrsim \mathcal{A}$. If $\zeta'' \neq \iota_{\mathsf{self}}$ then $\mathcal{A}^*(\zeta'') = \mathcal{A}'''(\zeta'')$ since $\mathcal{A}^* = \mathcal{A}'''[\iota_{\mathsf{self}} \mapsto \sigma(\mathsf{self})]$.

The proof of (2) shows also $\mathcal{A}^* \gtrsim \mathcal{A}'$: this together with $(\sigma, \sigma') \propto^{\mathcal{A}'} \varphi$ gives $(\sigma, \sigma') \propto^{\mathcal{A}^*} \varphi$ by Lemma 1(2). The proof of (3) shows also $\mathcal{A}'''(\iota_{\mathsf{self}}) = \mathcal{A}'(\zeta)$. From this, $\sigma'''(\mathsf{self}) = \mathcal{A}'(\zeta)$ and $(\sigma''', \sigma'') \propto^{\mathcal{A}'''} \varphi'$ we get
$(\sigma'''[\mathsf{self} \mapsto \sigma(\mathsf{self})], \sigma''[\mathsf{self} \mapsto \sigma(\mathsf{self})]) \propto^{\mathcal{A}^*} \varphi'[\zeta/\iota_{\mathsf{self}}]$
by Lemma 1(6). Now

$$\sigma'''[\mathsf{self} \mapsto \sigma(\mathsf{self})] = \sigma'[\mathsf{self} \mapsto \mathcal{A}'(\zeta)][\mathsf{self} \mapsto \sigma(\mathsf{self})] = \sigma'$$

Then we derive $(\sigma', \sigma''[\mathsf{self} \mapsto \sigma(\mathsf{self})]) \propto^{\mathcal{A}^*} \varphi'[\zeta/\iota_{\mathsf{self}}]$: this together with $(\sigma, \sigma') \propto^{\mathcal{A}^*} \varphi \circ \varphi'[\zeta/\iota_{\mathsf{self}}]$ gives (4) by definition.

We go on now with the case of lazy updating. In this case $\mathsf{a} \equiv \mathsf{a}'.\mathsf{m} \lhd \mathsf{b}$ and the last applied rule in the typing of a is (R-lazy-up):

$$\frac{\Gamma \mid \Delta \vdash \mathsf{a}' : \mathsf{Obj}(\zeta), \varphi \quad \mathcal{L}a(\varphi(\Gamma), \zeta, \mathsf{m}) = \zeta' \quad \Theta \vdash \mathsf{b} : \tau, \varphi'}{\Gamma \mid \Delta \vdash \mathsf{a}'.\mathsf{m} \lhd \mathsf{b} : \mathsf{Obj}(\zeta), \varphi \circ \{\{\zeta' : \langle\!\langle \mathsf{m} : \langle \Theta, \tau, \varphi' \rangle \rangle\!\rangle_{\bullet}\}\}}$$

Let us assume $\overline{\mathsf{a}'}.\mathsf{m} \lhd \overline{\mathsf{b}}$ does not diverge. Given σ and $\mathcal{A}$ such that $\sigma \propto^{\mathcal{A}} \Gamma$, what we need to prove is:

(5) $\overline{\mathsf{a}'}.\mathsf{m} \lhd \overline{\mathsf{b}}, \sigma \rightsquigarrow_{\delta} \mathcal{A}'(\zeta), \sigma'$ for some $\mathcal{A}'$, σ' such that $\mathcal{A}' \gtrsim \mathcal{A}$, and

$$(\sigma, \sigma') \propto^{\mathcal{A}'} \varphi \circ \{\{\zeta' : \langle\!\langle \mathsf{m} : \langle \Theta, \tau, \varphi' \rangle \rangle\!\rangle_{\bullet}\}\}$$

By the induction hypothesis, $\Gamma \mid \Delta \vdash \mathsf{a}' : \mathsf{Obj}(\zeta), \varphi$ and the hypotheses of the theorem imply that either $\overline{\mathsf{a}'}, \sigma$ diverges or $\overline{\mathsf{a}'}, \sigma \rightsquigarrow_{\delta} \mathcal{A}_1(\zeta), \sigma'$ for some $\mathcal{A}_1$, σ' such that $\mathcal{A}_1 \gtrsim \mathcal{A}$, and $(\sigma, \sigma') \propto^{\mathcal{A}_1} \varphi$. Only the second possibility needs to be considered since if $\overline{\mathsf{a}'}, \sigma$ diverges

then also $\overline{\mathsf{a}}, \sigma$ diverges.
Hence we get

$$\overline{\mathsf{a}'}.\mathsf{m} \lhd \overline{\mathsf{b}}, \sigma \ \leadsto_{\mathcal{S}} \ \mathcal{A}_1(\zeta), \sigma'\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \overline{\mathsf{b}}\}$$

by rule $(LazyUpdate)$.

By defining $\mathcal{A}' = \mathcal{A}_1$ and $\sigma' = \sigma'\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \mathsf{b}\}$, we can prove (5) if we manage to show that

$$(\sigma, \sigma'\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \overline{\mathsf{b}}\}) \propto^{\mathcal{A}_1} \varphi \circ \{\{\zeta' : \langle\!\langle \mathsf{m} : \langle \Gamma', \tau, \varphi' \rangle \rangle\!\rangle_\bullet\}\}$$

This, by definition of $\propto$, corresponds to showing that

(6) $\sigma \propto \mathcal{A}_1$;
(7) $\sigma'\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \overline{\mathsf{b}}\} \propto \mathcal{A}_1$;
(8) $(\sigma, \sigma'\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \overline{\mathsf{b}}\}) \widetilde{\propto}^{\mathcal{A}_1} \varphi \circ \{\{\zeta' : \langle\!\langle \mathsf{m} : \langle \Gamma', \tau, \varphi' \rangle \rangle\!\rangle_\bullet\}\}$.

Clause (6) descends immediately from the definition of $\propto$ and the fact that, as seen above, by the induction hypothesis, we have $(\sigma, \sigma') \propto^{\mathcal{A}_1} \varphi$ and hence $\sigma \propto \mathcal{A}_1$ and $\sigma' \propto \mathcal{A}_1$. From $\sigma' \propto \mathcal{A}_1$ it descends also clause (7), since the operation $\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \overline{\mathsf{b}}\}$ does not modify the address associated to self.
Then what we have to show is (8), that is, by definition, we have to show that

$$\exists \sigma''.(\sigma, \sigma'') \widetilde{\propto}^{\mathcal{A}_1} \varphi \ \& \ (\sigma'', \sigma'\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \overline{\mathsf{b}}\}) \widetilde{\propto}^{\mathcal{A}_1} \{\{\zeta' : \langle\!\langle \mathsf{m} : \langle \Gamma', \tau, \varphi' \rangle \rangle\!\rangle_\bullet\}\}$$

By taking $\sigma'' = \sigma'$ we obtain the first clause by what we have already inferred from the induction hypothesis. Hence one needs to show that

$$(\sigma', \sigma'\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \overline{\mathsf{b}}\}) \widetilde{\propto}^{\mathcal{A}_1} \{\{\zeta' : \langle\!\langle \mathsf{m} : \langle \Gamma', \tau, \varphi' \rangle \rangle\!\rangle_\bullet\}\}$$

that is, by definition, one needs to show that:

(9) $\forall \iota' \neq \mathcal{A}_1(\zeta').\sigma'(\iota') = \sigma'\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \overline{\mathsf{b}}\}(\iota')$
(10) $\forall \rho.\sigma' \propto^{\mathcal{A}_1} \{\zeta' : \rho\} \implies \sigma'\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \overline{\mathsf{b}}\} \propto^{\mathcal{A}_1} \{\zeta' : \langle\!\langle \mathsf{m} : \langle \Gamma', \tau, \varphi' \rangle \rangle\!\rangle_\bullet(\rho)\}$

Notice that $\sigma' \propto^{\mathcal{A}_1} \varphi(\Gamma)$ by Lemma 1(3) since $\sigma \propto^{\mathcal{A}} \Gamma$, $\mathcal{A}_1 \gtrsim \mathcal{A}$ and $(\sigma, \sigma') \propto^{\mathcal{A}_1} \varphi$. Moreover $\mathcal{L}a(\varphi(\Gamma), \zeta, \mathsf{m}) = \zeta'$: then (9) follows from Lemma 2(1).
To get (10) by definition of agreement of a store with an environment and by the induction hypothesis on $\Theta \vdash \mathsf{b} : \tau, \varphi'$ we can see that we need only to prove that:

$$\mathcal{L}ook(\sigma'\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \overline{\mathsf{b}}\}, \mathcal{A}(\zeta'), \mathsf{m}) = \overline{\mathsf{b}}$$

which is an immediate consequence of Lemma 2(2).

We end considering the case in which the last applied rule is (R-union-E):

$$\frac{\begin{array}{l}\Gamma \mid \Delta; x : \Gamma', \tau', \varphi' \vdash \mathsf{c} : \tau, \varphi \qquad \Gamma \mid \Delta; x : \Gamma', \tau'', \varphi' \vdash \mathsf{c} : \tau, \varphi \\ \Gamma' \mid \Delta \vdash \mathsf{b} : \mathsf{Obj}(\zeta'), \mathsf{id} \\ \Gamma' \mid \Delta \vdash \mathsf{a}' : \tau', \varphi' \qquad \Gamma' \mid \Delta \vdash \mathsf{a}'' : \tau'', \varphi'\end{array}}{\Gamma \mid \Delta \vdash \mathsf{c}[\mathsf{if}\ \mathsf{b}\ \mathsf{then}\ \mathsf{a}'\ \mathsf{else}\ \mathsf{a}''/x] : \tau, \varphi}$$

If $\overline{\mathsf{c}[\mathsf{if}\ \mathsf{b}\ \mathsf{then}\ \mathsf{a}'\ \mathsf{else}\ \mathsf{a}''/\mathsf{x}]}$ diverges we are done. If $\mathcal{S}(\overline{\mathsf{b}}, \overline{\mathsf{a}}, \sigma)$ is undefined then by Lemma 3:

$$\overline{\mathsf{c}[\mathsf{if}\ \mathsf{b}\ \mathsf{then}\ \mathsf{a}'\ \mathsf{else}\ \mathsf{a}''/\mathsf{x}]} \sim^{\sigma} \overline{\mathsf{c}[\mathsf{a}'/\mathsf{x}]}.$$

The typing judgement $\Gamma' \mid \Delta \vdash \mathsf{b} : \mathsf{Obj}(\zeta'), \mathsf{id}$ implies by induction that for all σ_b, $\mathcal{A}_b$, such that $\sigma_b \propto^{\mathcal{A}_b} \Gamma'$ either $\overline{\mathsf{b}}, \sigma_b$ diverges or $\overline{\mathsf{b}}, \sigma_b \leadsto_{\mathcal{S}} \mathcal{A}'_b(\zeta'), \sigma'_b$ for some $\mathcal{A}'_b$, σ'_b such that $\mathcal{A}'_b \gtrsim \mathcal{A}_b$, and $(\sigma_b, \sigma'_b) \propto^{\mathcal{A}'_b} \mathsf{id}$. By definition $(\sigma_b, \sigma'_b) \propto^{\mathcal{A}'_b} \mathsf{id}$ gives $\sigma'_b = \sigma_b$, i.e. the evaluation of b in a state which "agrees" with Γ' never changes the state.
From the typing judgement $\Gamma \mid \Delta; x : \Gamma', \tau', \varphi' \vdash \mathsf{c} : \tau, \varphi$ we get by induction on clause (1) (considering one fixed occurrence of x in c playing the role of b of clause (1)) that for all $\langle \Gamma', \tau', \varphi' \rangle$-sound δ^+ expressions e if $\mathcal{S}(\mathsf{e}, \overline{\mathsf{c}[\mathsf{e}/\mathsf{x}]}, \sigma)$ is defined then $\mathcal{S}(\mathsf{e}, \overline{\mathsf{c}[\mathsf{e}/\mathsf{x}]}, \sigma) \propto^{\mathcal{A}} \Gamma'$.
Being by above $\mathcal{S}(\mathsf{e}, \overline{\mathsf{c}[\mathsf{e}/\mathsf{x}]}, \sigma) = \mathcal{S}(\overline{\mathsf{b}}, \overline{\mathsf{a}}, \sigma) = \mathcal{S}(\overline{\mathsf{a}'}, \overline{\mathsf{a}}, \sigma) = \mathcal{S}(\overline{\mathsf{a}''}, \overline{\mathsf{a}}, \sigma)$, if they are defined then $\mathcal{S}(\overline{\mathsf{b}}, \overline{\mathsf{a}}, \sigma) \propto^{\mathcal{A}} \Gamma'$, $\mathcal{S}(\overline{\mathsf{a}'}, \overline{\mathsf{a}}, \sigma) \propto^{\mathcal{A}} \Gamma'$ and $\mathcal{S}(\overline{\mathsf{a}''}, \overline{\mathsf{a}}, \sigma) \propto^{\mathcal{A}} \Gamma'$: this together with induction gives clause (1).
Since $\mathcal{S}(\overline{\mathsf{b}}, \overline{\mathsf{a}}, \sigma) \propto^{\mathcal{A}} \Gamma'$ when defined, the evaluation of b does not change the state, so we can apply Lemma 3 also when $\mathcal{S}(\overline{\mathsf{b}}, \overline{\mathsf{a}}, \sigma)$ is defined and we obtain:

$$\overline{\mathsf{c}[\mathsf{if}\ \mathsf{b}\ \mathsf{then}\ \mathsf{a}'\ \mathsf{else}\ \mathsf{a}''/\mathsf{x}]} \sim^{\sigma} \overline{\mathsf{c}[\mathsf{d}/\mathsf{x}]}$$

where $\mathsf{d} \in \{\mathsf{a}', \mathsf{a}''\}$. We consider the case $\mathsf{d} \equiv \mathsf{a}'$, the other being the same. By induction $\Gamma' \mid \Delta \vdash \mathsf{a}' : \tau', \varphi'$ implies that $\overline{\mathsf{a}'}$ is $\langle \Gamma', \tau', \varphi' \rangle$-sound. Again by induction $\Gamma \mid \Delta; x : \Gamma', \tau', \varphi' \vdash \mathsf{c} : \tau, \varphi$ implies that $\overline{\mathsf{c}[\overline{\mathsf{a}'}/\mathsf{x}]}$ is $\langle \Gamma, \tau, \varphi \rangle$-sound. We conclude observing that $\overline{\mathsf{c}[\overline{\mathsf{a}'}/\mathsf{x}]} \equiv \overline{\mathsf{c}[\mathsf{a}'/\mathsf{x}]}$. ∎

IV. Conclusions and Future Work

As it usually happens in many type systems, our system rejects many expressions which correctly evaluate. In fact there is no polymorphism in the present system. We indeed plan to explore this issue in the line of [10].

In our system, even if it is possible to type diverging expressions like $\iota.\mathsf{m} \lhd \mathsf{self}.\mathsf{m}\ ;\ \iota.\mathsf{m}$, it can be noticed that types are very close to "computation traces". This might invite a rather severe criticism, and indeed the design of types abstracting more from the behaviour of expressions is definitely an issue we plan to work on in future investigations.

Acknowledgements

We would like to thank Sophia Drossopoulou, Paola Giannini, and Ferruccio Damiani for helpful discussions about the subject of this paper. Moreover we are grateful to the referees of WOOD'03 for careful reading and useful suggestions. The second author is also grateful to Alessandro Cavalli and Gisella Meli.

REFERENCES

[1] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, J. Maloney, Randall B. Smith, and David Ungar. The SELF Programmers's Reference Manual, version 2.0. Technical report, SUN Microsystems, 1992.

[2] Christopher Anderson, Franco Barbanera, Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou. Can Addresses be Types? (a case study: objects with delegation) In V. Bono and M. Bugliesi, editors, *WOOD'03*, Electronic Notes in Theoretical Computer Science. Elsevier, 82 No.8, 2003.

[3] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, Berlin, 1996.

[4] Christopher Anderson and Sophia Drossopoulou. δ - an Imperative Object Based Calculus. Presented at the workshop USE in 2002, Malaga, `http://www.binarylord.com/work/delta.pdf`, 2002.

[5] Kim Bruce. *Foundations of Object-Oriented Programming Languages: Types and Semantics*. The MIT Presse, Cambridge, MA, 2002.

[6] Ferruccio Damiani and Paola Giannini. Alias Types for "Environment-Aware" Computation. In V. Bono and M. Bugliesi, editors, *WOOD'03*, Electronic Notes in Theoretical Computer Science. Elsevier, 82 No.8, 2003.

[7] Kathleen Fisher and John Mitchell. A Delegation-Based Object Calculus with Subtyping. In H. Reichel, editor, *FCT'95*, volume 965 of *Lecture Notes in Computer Science*, pages 42–61. Springer-Verlag, Berlin, 1995.

[8] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In N. K. Meyrowitz, editor, *OOPSLA'86, ACM SIGPLAN Notices*, volume 21(11), pages 214–223, ACM Press, New York, NY, 1986.

[9] Benjamin C. Pierce. *Types and Programming Languages: Semantics*. The MIT Press, Cambridge, MA, 2002.

[10] Frederick Smith, David Walker, and Greg Morrisett. Alias Types. In G. Smolka, editor, *ESOP'00*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, Springer-Verlag, Berlin, 2000.

[11] David Walker and Greg Morrisett. Alias Types for Recursive Data Structures. In R. Harper, editor, *TIC'00*, volume 2071 of *Lecture Notes in Computer Science*, pages 117–146, Springer-Verlag, Berlin, 2001.