Electronic Notes in Theoretical Computer Science 82 No. 8 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 22 pages

Can Addresses be Types? (a case study: objects with delegation)

Christopher Anderson [1,4]

Department of Computing, Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, U.K.

Franco Barbanera [2,5]

Dipartimento di Matematica e Informatica, Università di Catania,
Viale A. Doria, 95125 Catania, Italy.

Mariangiola Dezani-Ciancaglini [3,6]

Dipartimento di Informatica, Università di Torino,
C.so Svizzera, 158, 10149 Torino, Italy.

Sophia Drossopoulou [1,7]

Department of Computing, Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, U.K.

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) whose 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 which supports method and delegate overriding, addition, and removing.

Key words: object based calculi, delegation, alias types, effects

[1] Work partly supported by DART, European Commission Research Directorates, IST-01-6-1A
[2] partially supported by MURST Cofin'01 project NAPOLI.
[3] partially supported by EU within the FET - Global Computing initiative, project DART ST-2001-33477, by MURST Cofin'01 project COMETA, 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.
[4] Email: cla97@doc.ic.ac.uk
[5] Email: barba@dmi.unict.it
[6] Email: dezani@di.unito.it
[7] Email: scd@doc.ic.ac.uk

1 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 [Lie86], [ABC$^+$92], [AD02].

A traditional approach for ensuring safety is typing. There is a large literature about calculi of objects with types (see [AC96a], [Bru02], [Pie02], 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.

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 [SWM01], [WM01] 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 objects with singleton types: the logical or physical addresses of the objects. The environments are 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, a simple intuitive calculus for imperative object based delegation [AD02].

It is worth mentioning that recently the same approach has been applied to a calculus for "environment-aware" computation [DG03].

The present paper is organized as follows: in Section 2 we introduce δ following [AD02]. Section 3 presents the type assignment system: types, typing rules and soundness proof.

2 The Calculus

We present δ, a minimal imperative object based calculus with delegation. Delegation has also been studied in [FM95] and its derivatives. Also in [AC96a] delega-

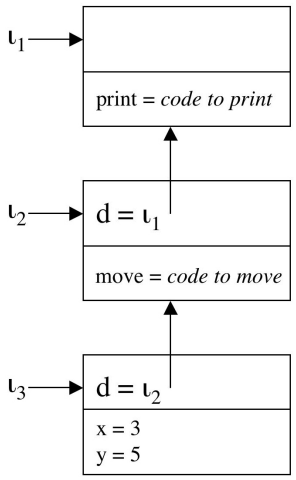

Fig. 1. Point with delegation

tion 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 [FM95] 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:

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

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

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

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 co-ordinates:

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

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

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

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

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

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`:

$$\iota.\mathsf{m} \lhd (\iota'.\mathsf{m}' \lhd \mathsf{self}) \; ; \; \iota'.\mathsf{m}', \sigma \; \leadsto_{\delta}^{\not\rightarrow} \; \texttt{stuckErr}, \sigma'$$

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

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

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.

Remark 2.1 In the previous examples we use parentheses which are not part of the syntax to help reading.

2.1 Syntax

The syntax (shown in figure 2) defines ten kinds of expressions: physical addresses, method invocation, lazy and eager update, clone, self, method removal, delegate invocation, addition and removal, composition. PhysAddress 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. We convene that $\mathsf{n} \in \mathsf{MethID} \uplus \mathsf{DelID}$, where $\uplus$ denotes union of disjoint sets.

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 $\mathsf{n} \in \mathsf{DelID}$.

2.2 Semantics

The operational semantics for δ is given in figure 5, 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 $\stackrel{\rightarrow}{fin}$ 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.

$$\rightsquigarrow_\delta \;:\; \mathsf{Exp} \times \mathsf{Store} \stackrel{\rightarrow}{fin} \mathsf{Address} \times \mathsf{Store}$$

$$\mathsf{Store} = (\{\mathsf{self}\} \to \mathsf{Address}) \cup (\mathsf{Address} \stackrel{\rightarrow}{fin} \mathsf{Obj})$$

$$\mathsf{Obj} = (\mathsf{MethID} \stackrel{\rightarrow}{fin} \mathsf{Exp}) \cup (\mathsf{DelID} \stackrel{\rightarrow}{fin} \mathsf{Address})$$

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 3. $\mathcal{L}ook'$ returns the set of pairs of addresses and bodies corresponding to a method identifier and an address in a given store. Lookup starts 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

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

Fig. 2. Syntax of δ

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 (*Eager Update*) and lazy (*Lazy Update*). 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 4. [8]

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

$$\mathcal{L}ook \;::\; \mathsf{Store} \times \mathsf{Address} \times (\mathsf{MethID} \uplus \mathsf{DelID}) \to \mathsf{Exp} \uplus \{\mathcal{U}df\}$$
$$\mathcal{L}ook' \;::\; \mathsf{Store} \times \mathsf{Address} \times (\mathsf{MethID} \uplus \mathsf{DelID}) \to \mathcal{P}(\mathsf{Exp} \times \mathsf{Address})$$

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

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

Fig. 3. 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. 4. Store Update and Remove

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

Clone (*Clone*) evaluates the object a then it allocates a new address and copies the object to the new address and returns the new address.

We define:

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

(*Self*)

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

(*Addr*)

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

(*Select*)

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

(*Delegate Select*)

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

(*Lazy Update*)

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

(*Eager Update*)

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

(*Delegate Update*)

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

(*Clone*)

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

(*Method Remove*)

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

(*Delegate Remove*)

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

(*Composition*)

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

(*Stuck Select*)

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

(*Stuck1 Delegate Select*)

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

(*Stuck2 Delegate Select*)

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

Fig. 5. Operational Semantics of δ

Method removal (*Method Remove*) and delegate removal (*Delegate Remove*) first evaluate the receiver then return the receiver with the delegate or method removed. The store removal, $\sigma\{\iota \rhd \mathsf{n}\}$, is defined in figure 4.

3 The Type Assignment System

Looking at the operational semantics of δ one easily sees that a `stuckErr` is generated only when a method invocation or a delegate invocation does not find a method or a delegate. 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. The typing judgements are of the shape:

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

where:

- the environment Γ gives informations 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 a;
- the type τ gives the address of the object, which is the value (if it exists) of the expression a;
- the effect φ gives the changes of the 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 of type τ and a store satisfying the constraints $\varphi(\Gamma)$, if the evaluation starts with a store satisfying Γ.

Typing a delegate update gives:

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

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:

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

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

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

Fig. 6. Types, Environments, Effects

3.1 Types, Environments, Effects, Judgments

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

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

Figure 6 lists the definitions of types, environments, and effects.
An *object type* is simply 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).
We have two kinds of *method types*.
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 informations about the typing of the bodies: if b has type $\langle \Gamma, \tau, \varphi \rangle$ then the judgment $\Gamma \vdash \mathsf{b} : \tau, \varphi$ 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.
Therefore 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.
Environments represent partial mappings between addresses and row types. They can be essentially seen as predicates (constraints) on stores.

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

Fig. 7. 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. 8. Effects

We use $\zeta \in \Gamma$ as short for $\exists \rho . \zeta : \rho \in \Gamma$.
A *row operator* is a total function from row types to row types as defined in figure 7. 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 environments to 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 8). 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 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 environment Γ only if $\zeta \notin \Gamma$. The dependent row overriding $\{\{\zeta : \psi\}\}$ must be applied to an environment containing a pair $\zeta : \rho$: the resulting environment will contain the pair $\zeta : \psi(\rho)$.

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

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

where Γ is an environment, τ is an object type and φ is an effect.

$$\frac{\iota \notin \Gamma}{\Gamma \vdash \iota : \mathsf{Obj}(\iota), \{[\iota : \langle\!\langle\ \rangle\!\rangle \parallel \langle\!\langle\ \rangle\!\rangle]\}}\ \text{(Ax-}\iota\text{-init)}$$

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

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

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

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

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

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

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

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

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

Fig. 9. Typing Rules

3.2 Typing Rules

Figure 9 lists the typing rules. We use the following notational convention:

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

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) add ι to the environment with the empty row. The other two axioms do not change the environment, so their effect is id.

Rule (R-clone) says that clone(a) has a fresh logical address and the row type of a. Notice that the row type of a is taken from the 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 environment obtained from the initial one by applying the effect of the typing of the first expression.

$$\mathcal{L}(\Gamma, \zeta, \mathsf{n}) = \begin{cases} \{(\lambda, \zeta)\} & \text{if } \langle\!\langle \mathsf{n} : \lambda \rangle\!\rangle \in \Gamma(\zeta) \\ \bigcup_{\zeta' \in \mathcal{I}^{\Gamma}(\zeta)} \mathcal{L}(\Gamma, \zeta', \mathsf{n}) & \text{otherwise} \end{cases}$$
$$\text{where } \mathcal{I}^{\Gamma}(\zeta) = \{\zeta' \mid \langle\!\langle \mathsf{d} : \mathsf{Obj}(\zeta') \rangle\!\rangle \in \Gamma(\zeta)\}$$

$$\mathcal{L}a(\Gamma, \zeta, \mathsf{n}) = \begin{cases} \zeta' & \text{if } \mathcal{L}(\Gamma, \zeta, \mathsf{n}) = \{(\lambda, \zeta')\} \\ \zeta & \text{otherwise} \end{cases}$$

$$\mathcal{L}t(\Gamma, \zeta, \mathsf{n}) = \begin{cases} \lambda & \text{if } \mathcal{L}(\Gamma, \zeta, \mathsf{n}) = \{(\lambda, \zeta')\} \\ \mathcal{U}df & \text{otherwise} \end{cases}$$

Fig. 10. The $\mathcal{L}$, $\mathcal{L}a$ and $\mathcal{L}t$ Functions

The method selection rules (R-eager-sel) and (R-lazy-sel) 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 10 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 10. The current 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 $\langle \Gamma', \tau, \varphi' \rangle$ 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 environment $\varphi(\Gamma)$ obtained from the initial one (Γ) by applying the effect φ of the typing of a. We know b is well typed in the environment Γ', so we have to check that $\varphi(\Gamma)$ is as good as Γ'. To this aim we introduce a partial order on environments. 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:

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

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 $\varphi \circ \{\![\iota_{\mathsf{self}} : \rho]\!\}(\Gamma) \leq \Gamma'$. 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 10. 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 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 7 and 8 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 Γ', which can have no relation with the environment Γ in which the receiver is typed. This is safe, since the type of the added method will recall Γ' 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 Γ'. If we can derive $\Gamma' \vdash \mathsf{b}:\tau,\varphi'$ then the method m will get the type $\langle\Gamma',\tau,\varphi'\rangle$; this type gives a complete information about the typing of b. The type of $\mathsf{a}.\mathsf{m} \lhd \mathsf{b}$ 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 7 and 8 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 type for the new delegate in the environment obtained from the initial one by applying the effect of the typing of the receiver. 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 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 7 and 8 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 resulting type is the type of the receiver. 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 7 the effects representing these deletions are respectively $\{\{\zeta : /_\bullet \mathsf{m}\}\}$ and $\{\{\zeta : /_@ \mathsf{d}\}\}$, where ζ is the address of the receiver.

3.3 Typing Example

We now give a detailed outline of the typing for the δ expression

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

given in section 2.

The overall typing is a series of applications of the rule (R-comp) (see Figure 11 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:

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

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

Fig. 11. Example of Typing

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

Fig. 12. Typing of $\iota'.\mathsf{m} \lhd (\iota''.\mathsf{m}' \blacktriangleleft \mathsf{self})$

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 environment:

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

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 environment Γ_1 we get:

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

The typing of $\Gamma_2 \vdash \iota'.\mathsf{m} \lhd (\iota''.\mathsf{m}' \blacktriangleleft \mathsf{self}) : \mathsf{Obj}(\iota'), \varphi_3$ (see Figure 12) uses rules (R-eager-up), (R-lazy-up) and we obtain effect:

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

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 $\langle \emptyset, \mathsf{Obj}(\iota''), \varphi_5 \rangle$.
Applying φ_3 to environment Γ_2 we get:

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

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:

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

Applying φ_4 to environment Γ_3 we get:

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

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.

3.4 Soundness

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

Fig. 13. Definition of $(\sigma, \sigma') \widetilde{\propto}^{\mathcal{A}} \varphi$

There are clear correspondences between the syntax of the calculus (Figure 2) and that of types (Figure 6), the look up functions of the operational semantics (Figure 3) and those of the typing rules (Figure 10), the store update (Figure 4) and the row operations (Figure 7). 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 $\mathcal{A}$.

We define agreement between stores, address mappings, effects and 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 environment Γ via the address mapping $\mathcal{A}$ iff the objects in the store have delegates and fields

$$
\begin{array}{lcl}
\mathcal{L}t(\Gamma,\zeta,\mathsf{n}) = \mathsf{Obj}(\zeta') & \Longrightarrow & \mathcal{L}ook(\sigma,\mathcal{A}(\zeta),\mathsf{n}) = \mathcal{A}(\zeta') \\
\mathcal{L}t(\Gamma,\zeta,\mathsf{m}) = \langle\Gamma',\mathsf{Obj}(\zeta'),\varphi\rangle & \Longrightarrow & \mathcal{L}ook(\sigma,\mathcal{A}(\zeta),\mathsf{m}) = \mathsf{b}\ \&\ \\
 & & \forall \mathcal{A}',\sigma' \text{ such that } \sigma' \propto^{\mathcal{A}'} \Gamma' \\
 & & \text{either } \mathsf{b},\sigma' \text{ diverges or} \\
 & & \mathsf{b},\sigma' \rightsquigarrow_{\delta} \mathcal{A}''(\zeta'),\sigma'' \text{ for some } \mathcal{A}'',\sigma'' \\
 & & \text{such that } \mathcal{A}'' \gtrsim \mathcal{A}', \text{ and } (\sigma',\sigma'') \propto^{\mathcal{A}} \varphi.
\end{array}
$$

Fig. 14. Definition of $\sigma \propto^{\mathcal{A}} \Gamma$

as required by the environment. Moreover the method bodies behave well when evaluated in environments which respect their typings.

Definition 3.1 (i) A *store σ agrees with an address mapping $\mathcal{A}$* (notation $\sigma \propto \mathcal{A}$) iff $\sigma(\mathsf{self}) = \mathcal{A}(\iota_{\mathsf{self}})$.

(ii) 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 13 (using the notion of point iv).

(iii) 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 $(\sigma,\sigma')\widetilde{\propto}^{\mathcal{A}}\varphi$.

(iv) A *store σ agrees with an 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 14.

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

Lemma 3.2 (i) *If $\sigma \propto^{\mathcal{A}} \Gamma$ and $\Gamma \leq \Gamma'$ then $\sigma \propto^{\mathcal{A}} \Gamma'$.*

(ii) *If $(\sigma,\sigma') \propto^{\mathcal{A}} \varphi$ and $\mathcal{A}' \gtrsim \mathcal{A}$ then $(\sigma,\sigma') \propto^{\mathcal{A}'} \varphi$.*

(iii) *If $(\sigma,\sigma') \propto^{\mathcal{A}} \varphi$ and $\sigma \propto^{\mathcal{A}} \Gamma$, then $\sigma' \propto^{\mathcal{A}} \varphi(\Gamma)$.*

(iv) *If $\mathcal{A}' \gtrsim \mathcal{A}$ then $\mathcal{A}'[\iota_{\mathsf{self}} \mapsto \iota] \gtrsim \mathcal{A}[\iota_{\mathsf{self}} \mapsto \iota]$.*

(v) *If $\sigma \propto^{\mathcal{A}} \Gamma$ and $\mathcal{A}' = \mathcal{A}[\iota_{\mathsf{self}} \mapsto \mathcal{A}(\zeta)]$ then $\sigma[\mathsf{self} \mapsto \mathcal{A}(\zeta)] \propto^{\mathcal{A}'} \{[\iota_{\mathsf{self}} : \Gamma(\zeta)]\}(\Gamma)$.*

(vi) *If $(\sigma,\sigma') \propto^{\mathcal{A}} \varphi$, $\mathcal{A}' = \mathcal{A}[\iota_{\mathsf{self}} \mapsto \iota]$, and $\sigma(\mathsf{self}) = \mathcal{A}(\zeta)$ then $(\sigma[\mathsf{self} \mapsto \iota], \sigma'[\mathsf{self} \mapsto \iota]) \propto^{\mathcal{A}} \varphi[\zeta/\iota_{\mathsf{self}}]$.*

Lemma 3.3 *Let $\sigma \propto^{\mathcal{A}} \Gamma$ and $\mathcal{L}a(\Gamma,\zeta,\mathsf{m}) = \zeta'$.*

(i) $\forall \iota \neq \mathcal{A}(\zeta').\sigma(\iota) = \sigma\{\mathcal{A}(\zeta).\mathsf{m} \lhd^{+} \mathsf{b}\}(\iota)$.

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

The type system is sound in the sense that a converging well typed expression, when evaluated in a suitable store, returns an address which agrees with the type of the expression, and never returns `stuckErr`.

Theorem 3.4 (Soundness) *If* $\Gamma \vdash \mathsf{a} : \mathsf{Obj}(\zeta), \varphi$ *and* $\sigma \propto^{\mathcal{A}} \Gamma$ *then either* $\mathsf{a}, \sigma \leadsto_{\delta}$ *diverges or* $\mathsf{a}, \sigma \leadsto_{\delta} \mathcal{A}'(\zeta), \sigma'$ *for some* $\mathcal{A}', \sigma'$ *such that* $\mathcal{A}' \gtrsim \mathcal{A}$, *and* $(\sigma, \sigma') \propto^{\mathcal{A}'} \varphi$.

Proof. The proof is by structural induction on a.
Let us consider the case when a is the selection of a delegate method added or overrided with a lazy updating. In this case $\mathsf{a} \equiv \mathsf{a}'@\mathsf{d}.\mathsf{m}$ and the last applied rule in the typing of a is rule (R-del-lazy-sel) (see Figure 15). By induction $\Gamma \vdash \mathsf{a}' : \mathsf{Obj}(\zeta), \varphi$

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

Fig. 15. Rule (R-del-lazy-sel)

implies that either a', σ diverges or $\mathsf{a}', \sigma \leadsto_{\delta} \mathcal{A}'(\zeta), \sigma'$ for some $\mathcal{A}'$, σ' such that $\mathcal{A}' \gtrsim \mathcal{A}$, and $(\sigma, \sigma') \propto^{\mathcal{A}'} \varphi$. If a', σ diverges then also a, σ diverges.
Otherwise by Lemma 3.2(iii) $(\sigma, \sigma') \propto^{\mathcal{A}} \varphi$ 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 $\mathcal{L}t(\varphi(\Gamma), \zeta', \mathsf{m}) = \langle \Gamma', \mathsf{Obj}(\zeta''), \varphi' \rangle$ we get $\mathcal{L}ook(\sigma, \mathcal{A}'(\zeta'), \mathsf{m}) = \mathsf{b}$ and for all $\mathcal{A}_b$, σ_b, such that $\sigma_b \propto^{\mathcal{A}_b} \Gamma'$ either b, σ_b diverges or $\mathsf{b}, \sigma_b \leadsto_{\delta} \mathcal{A}'_b(\zeta''), \sigma'_b$ 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 3.2(v) $\sigma' \propto^{\mathcal{A}'} \varphi(\Gamma)$ 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 3.2(i). Then either b, σ''' diverges or $\mathsf{b}, \sigma''' \leadsto_{\delta} \mathcal{A}'''(\zeta''), \sigma''$ for some $\mathcal{A}'''$, σ'' such that $\mathcal{A}''' \gtrsim \mathcal{A}''$, and $(\sigma''', \sigma'') \propto^{\mathcal{A}'''} \varphi'$. If b, σ''' diverges then also a, σ diverges.
If $\mathsf{b}, \sigma''' \leadsto_{\delta} \mathcal{A}'''(\zeta''), \sigma''$ then $\mathsf{a}, \sigma \leadsto_{\delta} \mathcal{A}'''(\zeta''), \sigma''[\mathsf{self} \mapsto \sigma(\mathsf{self})]$.
Let $\mathcal{A}^* = \mathcal{A}'''[\iota_{\mathsf{self}} \mapsto \sigma(\mathsf{self})]$: then it suffices to show:

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

For (1) 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 3.2(iv). 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 (2) 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 accont 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 (1) 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 3.2(ii). The proof of (2) 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 3.2(vi).
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 (3) 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:

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

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

(4) $\mathsf{a}'.\mathsf{m} \lhd \mathsf{b}, \sigma \rightsquigarrow \mathcal{A}'(\zeta), \sigma'$ for some $\mathcal{A}'$, σ' such that
$\mathcal{A}' \gtrsim \mathcal{A}$, and $(\sigma, \sigma') \propto^{\mathcal{A}'} \varphi \circ \{\{\zeta' : \langle\!\langle \mathsf{m} : \langle \Gamma', \tau, \varphi' \rangle \rangle\!\rangle_\bullet\}\}$.

By the induction hypothesis, $\Gamma \vdash \mathsf{a}' : \mathsf{Obj}(\zeta), \varphi$ and the hypotheses of the theorem imply that either a', σ diverges or $\mathsf{a}', \sigma \rightsquigarrow \mathcal{A}_1(\zeta), \sigma_1$ for some $\mathcal{A}_1$, σ_1 such that $\mathcal{A}_1 \gtrsim \mathcal{A}$, and $(\sigma, \sigma_1) \propto^{\mathcal{A}_1} \varphi$. Only the second possibility need to be considered since if a', σ diverges then also a, σ diverges.
Hence, by rule $(LazyUpdate)$, we get $\mathsf{a}'.\mathsf{m} \lhd \mathsf{b}, \sigma \rightsquigarrow \mathcal{A}_1(\zeta), \sigma_1\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \mathsf{b}\}$.

By defining $\mathcal{A}' = \mathcal{A}_1$ and $\sigma' = \sigma_1\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \mathsf{b}\}$, we can prove (4) if we manage to show that

$$(\sigma, \sigma_1\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \mathsf{b}\}) \propto^{\mathcal{A}_1} \varphi \circ \{\{\zeta' : \langle\!\langle \mathsf{m} : \langle \Gamma', \tau, \varphi' \rangle \rangle\!\rangle_\bullet\}\}$$

This, by definition of $\propto$, corresponds to showing that $\sigma \propto \mathcal{A}_1$, $\sigma_1\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \mathsf{b}\} \propto \mathcal{A}_1$ and $(\sigma, \sigma_1\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \mathsf{b}\}) \widetilde{\propto}^{\mathcal{A}_1} \varphi \circ \{\{\zeta' : \langle\!\langle \mathsf{m} : \langle \Gamma', \tau, \varphi' \rangle \rangle\!\rangle_\bullet\}\}$.

The first clause descends immediately from the definition of $\propto$ and the fact that, as seen above, by the induction hypothesis, we have $(\sigma, \sigma_1) \propto^{\mathcal{A}_1} \varphi$ and hence $\sigma \propto \mathcal{A}_1$ and $\sigma_1 \propto \mathcal{A}_1$. From $\sigma_1 \propto \mathcal{A}_1$ it descends also the second clause, since the operation $\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \mathsf{b}\}$ does not modify the address associated to self.
Then what we have to show is

$$(\sigma, \sigma_1\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \mathsf{b}\}) \widetilde{\propto}^{\mathcal{A}_1} \varphi \circ \{\{\zeta' : \langle\!\langle \mathsf{m} : \langle \Gamma', \tau, \varphi' \rangle \rangle\!\rangle_\bullet\}\}$$

that is, by definition, we have to show that

$$\exists\sigma''.(\sigma,\sigma'')\widetilde{\propto}^{\mathcal{A}_1}\varphi \;\&\; (\sigma'',\sigma_1\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \mathsf{b}\})\widetilde{\propto}^{\mathcal{A}_1}\{\{\zeta' : \langle\!\langle \mathsf{m} : \langle\Gamma',\tau,\varphi'\rangle\rangle\!\rangle_\bullet\}\}$$

By taking $\sigma'' = \sigma_1$ we obtain the first clause by what we have already inferred from the induction hypothesis. Hence one needs to show that

$$(\sigma_1,\sigma_1\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \mathsf{b}\})\widetilde{\propto}^{\mathcal{A}_1}\{\{\zeta' : \langle\!\langle \mathsf{m} : \langle\Gamma',\tau,\varphi'\rangle\rangle\!\rangle_\bullet\}\}$$

that is, by definition, one needs to show that

(5) $\forall\iota' \neq \mathcal{A}_1(\zeta').\sigma_1(\iota') = \sigma_1\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \mathsf{b}\}(\iota')$

(6) $\forall\rho.\sigma_1 \propto^{\mathcal{A}_1} \{\zeta' : \rho\} \implies \sigma_1\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \mathsf{b}\} \propto^{\mathcal{A}_1} \{\zeta' : \langle\!\langle \mathsf{m} : \langle\Gamma',\tau,\varphi'\rangle\rangle\!\rangle_\bullet(\rho)\}$

Notice that $\sigma_1 \propto^{\mathcal{A}_1} \varphi(\Gamma)$ by Lemma 3.2(iii) since $\sigma \propto^{\mathcal{A}} \Gamma$, $\mathcal{A}_1 \gtrsim \mathcal{A}$ and $(\sigma,\sigma_1) \propto^{\mathcal{A}_1} \varphi$. Moreover $\mathcal{L}a(\varphi(\Gamma),\zeta,\mathsf{m}) = \zeta'$: then (5) follows from Lemma 3.3(i).
To get (6) by definition of agreement of a store with an environment and by the induction hypotesis on $\Gamma' \vdash \mathsf{b} : \tau,\varphi'$ we can see that we need only to prove that

$$\mathcal{L}ook(\sigma_1\{\mathcal{A}_1(\zeta).\mathsf{m} \lhd^+ \mathsf{b}\},\mathcal{A}(\zeta'),\mathsf{m}) = \mathsf{b}$$

which is an immediate consequence of Lemma 3.3(ii).

□

4 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 [SWM01].

As noted by two referees the extension of the present type system to conditional expressions is not trivial. As a matter of fact [DG03] gives a system of alias types for an object based calculus with conditionals and a test for the absence/presence of methods. Of course the delegation complicates the matter significantly, but we already sketched some proposals we have to put at work on meaningful examples.

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 Paola Giannini, and Ferruccio Damiani for helpful discussions about the subject of this paper. Moreover we are grateful to the referees for

careful reading and useful suggestions.

References

[ABC+92] 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.

[AC96a] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, Berlin, 1996.

[AD02] 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.

[Bru02] Kim Bruce. *Foundations of Object-Oriented Programming Languages: Types and Semantics*. The MIT Presse, Cambridge, MA, 2002.

[DG03] Ferruccio Damiani and Paola Giannini. Alias types for "environment-aware" computation. In *WOOD'03*, Electronic Notes in Theoretical Computer Science. Elsevier, 82 No.8, 2003.

[FM95] Kathleen Fisher and John Mitchell. A delegation-based object calculus with subtyping. In *FCT'95*, volume 965 of *Lecture Notes in Computer Science*, pages 42–61. Springer-Verlag, Berlin, 1995.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPSLA'86, ACM SIGPLAN Notices*, volume 21(11), pages 214–223, ACM Press, New York, NY, 1986.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages: Semantics*. The MIT Presse, Cambridge, MA, 2002.

[SWM01] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *ESOP'00*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, Springer-Verlag, Berlin, 2000.

[WM01] David Walker and Greg Morrisett. Alias types for recursive data structures. In *TIC'01*, volume 2071 of *Lecture Notes in Computer Science*, pages 117–146, Springer-Verlag, Berlin, 2001.