δ: an imperative object based calculus with delegation *

Christopher Anderson, Sophia Drossopoulou

Imperial College of Science, Technology and Medicine

Abstract. Object based, imperative languages with delegation (eg SELF) support exploratory programming: composition of objects, sharing of attributes and modification of objects' behaviour at run-time are easily expressible. Delegation allows objects to delegate execution of methods to other objects and to adapt delegated behaviour by overriding of method definitions. These features allow for creation of very flexible programs that can accomodate requirement changes at a very late stage.
Programming language design and understanding has generally benefited from formal models of programming languages. Such models, of course, tend to focus on the essential features of the particular language, making it possible to highlighting design alternatives and compare them. Models for object based languages have already been developed in the 90's [1, 7, 3], but these models do not directly express imperative delegation: In [1] delegation can only be encoded, while [7] is functional and its imperative derivative [3] does not accurately reflect the effect of change of a delegate. We argue that no calculi so far developed fully express the essence of such languages. We give a simple intuitive calculus for imperative object based delegation. We start with δ^-, an imperative object based calculus, and demonstrate its use through examples. $\delta^-$is similar to the first order imperative Abadi Cardelli calculus, though simpler in some sense. We prove a correspondence theorem. We then present δ, which extends δ^-through explicit delegation; δ allows an object to specify explicitly to which objects it wants to delegate. We show key features of δ through examples.

1 Introduction

Object based programming supports "exploratory programming", i.e. the prototyping of ideas through the construction of prototypical objects that are composed together to obtain a working system.

Delegation, [12], is a language feature which supports development of prototypical systems. Delegation can be used to represent class based programming in an object based environment more elegantly than pre-methods [1]. Languages

* Work partly supported by DART, European Commission Research Directorates, IST-01-6-1A

like SELF[6] allow an object to delegate execution of a method to another object. Delegation has recently been added as a feature to statically typed class based languages [11, 14, 15].

The dynamic features present in object based languages, such as method update and addition *coupled* with delegation give the ability to quickly react to changes in requirement even *after* execution has started, and thus support unanticipated evolution.

We develop a succinct calculus, δ, for such an object based language with delegation. In section 3 we present δ^-, give examples, and compare δ^- to the Abadi-Cardelli ς-calculus. In section 2 we describe delegation in more detail as a language feature. In section 4 we present δ.

A longer document containing more explanations, motivation, the complete formalism and parts of proofs can be found at [2].

2 Object Based Programming with Delegation for Unanticipated Software Evolution

The ability to update methods allows objects to change their behaviour at run-time, and thus to adapt to change of requirements, while the system is running. For example, consider a setting with black and white printers, and corresponding print methods. When moving to a setting with colour printers, all that needs to be done is to update the print method, which in our syntax can be done as follows: $\iota.\mathsf{print} \lhd \mathit{ColourPrintCode}$. This can be done *without* shutting down the system, and *without* recompiling the whole code.

Furthermore, adding new methods at run-time allows the objects to extend their functionality as the requirements are extended. For example, consider a setting where we have two dimensional figures, and we want to move them to a three dimensional setting. All that needs to be done is to add the third dimension, through a method z to the points defining the figures, *i.e.* $\iota.\mathsf{z} \lhd 5$. Again, this can be done *without* shutting down the system, and *without* recompiling the whole code.

Delegation in a programming language context was introduced by Lieberman [12] within the framework of a prototypical object based language: a delegator object can have mutable references to delegate objects. Lieberman called the delegator and delegate objects child and parent respectively. When a message sent to a delegator is not understood, it is automatically forwarded to its delegates. This process continues until a suitable method is found, upon which the original message receiver is bound to the implicit self parameter of the delegate. Hence, the delegate executes the method on behalf of the original message receiver. Object based languages that include delegation include:

- SELF - All objects have a parent or delegate *slot* (the term slot is a synonym of our method). Objects may have one or more parent slots. During message lookup if the search fails in the current object it continues with the parent slots.

– Act-1 [13] - A language based on Actors (active objects) that communicate via message passing. When an actor cannot process a message he delegates to a proxy (the delegate).
– Fisher and Mitchell [7] have modelled delegation in an object based calculus using copying.

There have also been attempts to add delegation to class based languages like Java:

– Kniesel [9] gives requirements for simulating delegation in Java using an API.
– Costanza and Kniesel [4, 10] added delegation to Java in a typesafe manner with the addition of a new keyword delegate.
– Viega [18] proposed adding delegation to Java in the context of allowing multiple inheritance.
– Ostermann [14] adds delegation and virtual inner classes to a simple class based language to allow the expression of a slice of behavior affecting a set of collaborating classes.

With delegation we can represent a point by *three* objects. One object that knows how to print points, another how to move points, and finally an object with x and y coordinates. Figure 1(a) 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 a_3 receives a move message it delegates it to a_2. Similarly, when a_3 receives a print message it delegates it to a_2 and a_2 delegates it to a_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 a_1 this would affect the behaviour of both a_2 and a_3.

Method sharing not only supports sharing of code, as shown earlier, but also sharing of state. Consider a vertical line segment with two point objects to mark its start and end. We now extend our example to include two points to make a line segment, by cloning a_3. Because delegation is transparent to object creation, the list of delegates in a_3 will be copied into the clone, cf figure 1(b). The segment is vertical, therefore both points will have the same x co-ordinate. This would best be represented by sharing of state. Figure 1(c) shows points a_3 and a_4 sharing the same x co-ordinate. When a_3 or a_4 are asked for their x co-ordinate they will delegate to a_2. Therefore, moving the vertical line segment left or right is simply a case of changing the *single* x co-ordinate in a_2 which in turn will affect *both* a_3 and a_4.

As already known, delegation can concisely represent class membership and inheritance [12, 16]. For example, consider objects that represent bank accounts which may be either savings accounts or current accounts. Using delegation, we can package the methods specific to current accounts into an object CurrentAccount and those specific to saving accounts into an object SavingsAccount, and we can package *their* common methods into an object Account. Particular accounts object will delegate to either SavingsAccount or CurrentAccount for their behaviour.

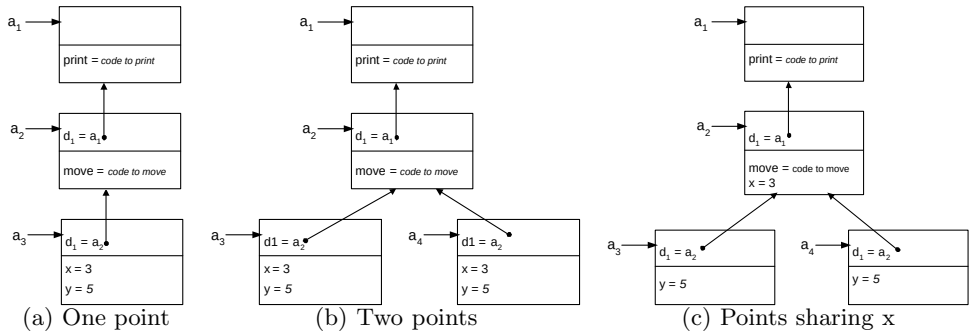

Fig. 1. Points with delegation

Delegation can be used to model *radical* change of behaviour; for example, we can turn a current account into a savings account by simply changing which object it delegates to. This has already been suggested in a class based setting in [10] where objects change delegates, and also in [5] where objects change class. However, in both above approaches the evolution is in some sense anticipated, because the class of the new delegate in [10], and the new class of the object in [5], are part of the program, and thus need to have been anticipated.

In an object based setting, delegation supports more forms of software evolution, *i.e.* change of behaviour for a *group* of objects, and unanticipated change of behaviour.

Changes of behaviour for a group of objects can be expressed through modification of a delegate object's methods. Consider, for example, a change to the method for calculating interest on savings accounts. This can easily be described through an update to methods in the object SavingsAccount. This will affect all objects which delegate to the SavingsAccount object. [1]

The unanticipated change of behaviour is expressed by creating a new object, which is then made a delegate to existing objects. Consider, for example, the situation where a new kind of account is created, say a super savings account. This can be described through the creation of a new object, say SuperSavingsAccount; any account object can be turned into a savings account by setting its delegate to be SuperSavingsAccount.

All the above can be done *without* shutting down the system, *without* recompilation, and while the system is running.

3 δ^-: An Imperative Object Calculus

We present δ^-, a minimal imperative object based calculus, which supports modification, addition and removal of methods from objects [8].

[1] Note, that this latter capability is not adequately reflected in the Fisher Mitchell calculus and its derivatives, e.g. [3], nor in the imperative ς-calculus.

The syntax (shown in figure 2) defines six kinds of expression: addresses, method invocation, lazy and eager update, clone, and self. Method identifiers range over an infinite set of names, MethID.

For example, starting with an empty object at address ι_1, we can create a point object as follows:

$$(((\iota_1.\mathsf{x} \blacktriangleleft 5).\mathsf{y} \blacktriangleleft 3).\mathsf{move} \triangleleft (\mathsf{self}.\mathsf{x} \blacktriangleleft \mathsf{self}.\mathsf{x} + 1)).\mathsf{print} \triangleleft \mathit{PrintCode}$$

Now the object at address ι_1 has four methods: print, move, x and y. The method x returns 3, the method y returns 5, the method move overwrites the method x, so that it returns the contents of x augmented by 1. We abbreviate the body of method print by *PrintCode*. Note, that $\iota_1.\mathsf{x} \blacktriangleleft 5$ denotes an eager update i.e. the argument (here 5) is evaluated and updates method x in ι_1. Whereas $(\iota_1...).\mathsf{move} \triangleleft (\mathsf{self}.\mathsf{x} \blacktriangleleft \mathsf{self}.\mathsf{x} + 1)$ denotes lazy update ie. the argument $(\mathsf{self}.\mathsf{x} \blacktriangleleft \mathsf{self}.\mathsf{x} + 1)$ remains unevaluated and as such overwrites the move method of the object at ι_1. For simplicity, we use the literals 1,2 ... as a shorthand for the object representations of the corresponding numbers. We do not have fields in our objects, but as in [1] we can encode them by eagerly evaluating the contents before update. This is demonstrated in the example above with the eager update of methods x and y with 3 and 5 respectively.

$$\begin{array}{rll}
\iota \in \mathsf{Address} & & \\
\mathsf{m} \in \mathsf{MethID} & & \\
\\
\mathsf{a}, \mathsf{b} \in \mathsf{Exp} ::= & \iota & \text{address} \\
& \mathsf{a}.\mathsf{m} & \text{method invocation} \\
& \mathsf{a}.\mathsf{m} \blacktriangleleft \mathsf{b} & \text{eager update} \\
& \mathsf{a}.\mathsf{m} \triangleleft \mathsf{b} & \text{lazy update} \\
& \mathsf{clone}(\mathsf{a}) & \text{clone (shallow)} \\
& \mathsf{self} & \text{receiver}
\end{array}$$

Fig. 2. Syntax of δ^-

The operational semantics for δ^-, is given in figure 3. It rewrites pairs of expression and stores into pairs of addresses and stores. The stores map addresses to objects and self to an address, where addresses are $\iota_0, ..\iota_n...$ Objects are finite mappings from method names to expressions, indicated by $\xrightarrow[fin]{}$.

$$\begin{array}{rcl}
\rightsquigarrow_\delta & : & \mathsf{Exp} \times \mathsf{Store} \xrightarrow[fin]{} \mathsf{Address} \times \mathsf{Store} \\
\mathsf{Store} & = & (\{\mathsf{self}\} \to \mathsf{Address}) \cup (\mathsf{Address} \xrightarrow[fin]{} \mathsf{Obj}) \\
\mathsf{Obj} & = & (\mathsf{MethID} \xrightarrow[fin]{} \mathsf{Exp})
\end{array}$$

We represent objects through $\mathsf{o} \equiv [\![\mathsf{m}_1 = \mathsf{b}_1 ... \mathsf{m}_n = \mathsf{b}_n]\!]$ denoting an object with methods $\mathsf{m}_1 ... \mathsf{m}_n$ and associated bodies $\mathsf{b}_1 ... \mathsf{b}_n$. We define store update,

$\sigma' = \sigma\{\iota.\mathsf{m} \lhd \mathsf{b}\}$, where σ' is identical to σ except that in ι method m is overridden/added by b:

$$\sigma\{\iota.\mathsf{m} \lhd \mathsf{b}\}(\iota)(\mathsf{m}) = \mathsf{b}, \quad \sigma\{\iota.\mathsf{m} \lhd \mathsf{b}\}(\iota)(\mathsf{m}') = \sigma(\iota)(\mathsf{m}') \ \text{ if } \mathsf{m}' \neq \mathsf{m}$$

The receiver self (*Self*) evaluates to the address of the receiver in the store. Many programming languages have a keyword for the receiver, e.g. this in Java and C++, self in Smalltalk and SELF.

Addresses $\iota_1..\iota_n$ (*Addr*) are part of the syntax and play, in some sense, the role of variables. Local definitions are not directly supported by the syntax, but can be represented through fields.

Method invocation (*Select*) starts by evaluating the receiver a to give an address ι. The body b of method m is looked up in the object at address ι, and b is evaluated in the context of a store where self is bound to the receiver ι. Finally, self is set back to the address it had before the method invocation.

We have two flavours of update: eager (*Eager Update*) and lazy (*Lazy Update*). Both evaluate the receiver a and return the address of the modified receiver. They differ in their treatment of b. Lazy update replaces the method body identified by m with the unevaluated body b. Eager update differs in that the body b is first evaluated before the update occurs. Hence, lazy update is like method update and eager update is like field update.

Clone (*Clone*) evaluates a then allocates a new address and copies the object to the new address and returns the new address. The copy is 'textual', giving a shallow copy of the receiver.

(*Self*)

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

(*Addr*)

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

(*Select*)

$$\frac{\begin{array}{l}\mathsf{a}, \sigma \ \leadsto_\delta \ \iota, \sigma' \\ \sigma'(\iota) = [\![...\mathsf{m} = \mathsf{b}...]\!] \\ \sigma''' = \sigma'[\mathsf{self} \mapsto \iota] \\ \mathsf{b}, \sigma''' \ \leadsto_\delta \ \iota', \sigma''\end{array}}{\mathsf{a.m}, \sigma \ \leadsto_\delta \ \iota', \sigma''[\mathsf{self} \mapsto \sigma(\mathsf{self})]}$$

(*Clone*)

$$\frac{\begin{array}{l}\mathsf{a}, \sigma \ \leadsto_\delta \ \iota, \sigma' \\ \iota' \notin dom(\sigma') \\ \sigma'' = \sigma'[\iota' \mapsto \sigma'(\iota)]\end{array}}{\mathsf{clone(a)}, \sigma \ \leadsto_\delta \ \iota', \sigma''}$$

(*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*)

$$\frac{\begin{array}{l}\mathsf{a}, \sigma \ \leadsto_\delta \ \iota, \sigma' \\ \mathsf{b}, \sigma' \ \leadsto_\delta \ \iota', \sigma''\end{array}}{\mathsf{a.m} \blacktriangleleft \mathsf{b}, \sigma \ \leadsto_\delta \ \iota, \sigma''\{\iota.\mathsf{m} \lhd \iota'\}}$$

Fig. 3. Operational semantics of δ^-

4 δ: An Imperative Object Calculus with Delegation

We present δ, an extension of δ^- with explicit delegation; method removal, delegate invocation, addition and removal. Delegation has also been studied in [7] and its derivatives. Also in [1] 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 $\mathsf{o_1}$ delegates to another object $\mathsf{o_2}$, then $\mathsf{o_2}$'s method body for m is modified or added and subsequent method call of m on $\mathsf{o_1}$ results in execution of the modified method body rather than the original one.

The syntax of δ (shown in figure 4) is that of δ^- extended with four new kinds of expression: method removal, delegate invocation, addition and removal. Method and delegate identifiers are taken from the disjoint infinite sets of names MethID and DelID respectively.

$$
\begin{array}{rll}
\iota \in \mathsf{Address} & & \\
\mathsf{m} \in \mathsf{MethID} & & \\
\mathsf{d} \in \mathsf{DelID} & & \\
\multicolumn{3}{l}{\quad \mathsf{MethID} \cap \mathsf{DelID} = \emptyset} \\
 & & \\
\mathsf{a}, \mathsf{b} \in \mathsf{Exp} ::= & \ldots & \text{as in figure 2} \\
 & \mathsf{a@d.m} & \text{delegate invocation} \\
 & \mathsf{a@d} \blacktriangleleft \mathsf{b} & \text{delegate update} \\
 & \mathsf{a} \rhd_{\bullet} \mathsf{m} & \text{method remove} \\
 & \mathsf{a} \rhd_{@} \mathsf{d} & \text{delegate remove}
\end{array}
$$

Fig. 4. Syntax of δ

In figure 7 we give a structural operational semantics similar to that of δ^-'s. In δ, as in δ^-, the stores map addresses to objects and self to an address, where addresses are $\iota_0..\iota_n...$ However, δ objects are "richer" than δ^- objects; they contain a list of delegates as well as a list of methods.

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

We represent objects through $\mathsf{o} \equiv [\![\mathsf{d_1} = \iota_1 ... \mathsf{d_k} = \iota_k \quad \| \quad \mathsf{m_1} = \mathsf{b_1} ... \mathsf{m_n} = \mathsf{b_n}]\!]$ denoting an object with delegates $\mathsf{d_1}...\mathsf{d_k}$ with associated addresses $\iota_1...\iota_k$ and methods $\mathsf{m_1}...\mathsf{m_n}$ with associated bodies $\mathsf{b_1}...\mathsf{b_n}$.

The example from section 2, can be represented in δ as:

$$\sigma_0(\iota_1) = [\![\quad \| \quad \mathsf{print} = \mathit{PrintCode}]\!]$$
$$\sigma_0(\iota_2) = [\![\mathsf{d}_{\mathsf{print}} = \iota_1 \quad \| \quad \mathsf{move} = \mathit{MoveCode}]\!]$$
$$\sigma_0(\iota_3) = [\![\mathsf{d}_{\mathsf{move}} = \iota_2 \quad \| \quad \mathsf{x} = 3, \mathsf{y} = 5]\!]$$

The rewrite rules are given in figure 7. We define store removal, $\sigma' = \sigma\{\iota \rhd \mathsf{n}\}$, where σ' is identical to σ except that method (or delegate) n is removed from ι, where $\mathcal{U}df$ means undefined:

$$\sigma\{\iota \rhd \mathsf{n}\}(\iota)(\mathsf{n}) = \mathcal{U}df,$$
$$\sigma\{\iota \rhd \mathsf{n}\}(\iota)(\mathsf{n}') = \sigma(\iota)(\mathsf{n}') \text{ if } \mathsf{n}' \neq \mathsf{n},$$
$$\sigma\{\iota \rhd \mathsf{n}\}(\iota') = \sigma(\iota') \text{ if } \iota \neq \iota'$$

The rewrite rules (*Select*) and (*Delegate Select*) use lookup functions $\mathcal{L}ook$ and $\mathcal{L}ook'$ shown in figure 5. $\mathcal{L}ook'$ 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 or delegate address is found, then the search continues in the delegates. Note that rules (*Select*) and (*Delegate Select*) require exactly one method or delegate body to be found through $\mathcal{L}ook$. 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 is stuck.

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

$$\mathcal{L}ook(\sigma, \iota, \mathsf{n}) = \{\mathsf{b} \mid (\mathsf{b}, \iota') \in \mathcal{L}ook'(\sigma, \iota, \mathsf{n}) \text{ for some } \iota'\}$$

$$\mathcal{L}ook'(\sigma, \iota, \mathsf{n}) = \begin{cases} \{(\mathsf{b}, \iota)\} & \text{if } \sigma(\iota) = [\![\ldots \quad \| \quad \ldots\ \mathsf{n} = \mathsf{b} \ldots]\!] \text{ or} \\ & \quad \sigma(\iota) = [\![\ldots\ \mathsf{n} = \iota' \ldots \quad \| \quad \ldots]\!], \mathsf{b} = \iota' \\ \bigcup_{\iota' \in I} \mathcal{L}ook'(\sigma, \iota', \mathsf{n}) & \text{otherwise} \end{cases}$$
$$\text{where } I = \{\iota_i \mid \sigma(\iota) = [\![\mathsf{d_i} = \iota_i{}^{i \in 1..n} \quad \| \quad \ldots]\!]\}$$

Fig. 5. The $\mathcal{L}ook$ and $\mathcal{L}ook'$ functions

Consider the following applications of $\mathcal{L}ook$ to state σ:

$$\mathcal{L}ook(\sigma_0, \iota_3, \mathsf{x}) = \{3\}$$
$$\mathcal{L}ook(\sigma_0, \iota_3, \mathsf{move}) = \{\mathit{MoveCode}\}$$
$$\mathcal{L}ook(\sigma_0, \iota_2, \mathsf{print}) = \{\mathit{PrintCode}\}$$
$$\mathcal{L}ook(\sigma_0, \iota_2, \mathsf{d}_{\mathsf{print}}) = \{\iota_1\}$$

Variations of the definition of $\mathcal{L}ook$ would give different behaviour, for example, we could model C++ multiple inheritance. The definition of $\mathcal{L}ook$ given here gives the same behaviour as message lookup in SELF ([6] defines a similar lookup algorithm).

In δ^- store update, $\sigma\{\iota.\mathsf{m} \lhd \mathsf{b}\}$, always updates or adds the method in the object at the specified address ι. We now define delegate store update, $\sigma\{\iota.\mathsf{n} \lhd^+ \mathsf{b}\}$, which we use in reduction rules *(Lazy Update), (Eager Update), (Delegate Update)*. Delegate store update uses $\mathcal{L}ook'$ to find the address of the object containing the specified method or delegate, starting the lookup from ι.

$$\sigma\{\iota.\mathsf{n} \lhd^+ \mathsf{b}\}(\iota')(\mathsf{n}') = \begin{cases} \sigma(\iota')(\mathsf{n}') & \text{if } \mathsf{n} = \mathsf{n}' \text{ or } (\iota' \neq \iota \text{ and } (\iota', \_) \notin \mathcal{L}ook'(\sigma, \iota, \mathsf{n})) \\ \mathsf{b} & \text{if } \mathsf{n} = \mathsf{n}', \iota' = \iota \text{ and } \mathcal{L}ook'(\sigma, \iota, \mathsf{n}) = \emptyset \\ \mathsf{b} & \text{if } \mathsf{n} = \mathsf{n}' \text{ and } (\iota', \_) \notin \mathcal{L}ook'(\sigma, \iota, \mathsf{n}) \end{cases}$$

Fig. 6. Delegate Store Update

Therefore, if an object ι has a method body for m, then delegate store update will replace that method body. If ι does not have a method body for m, but has delegates ι_1, ι_2, ι_3 and ι_1 and ι_3 have a method body for m, then store update will replace the methods in ι_1 and ι_3. Similarly for the update of delegates.

In terms of the points example, consider points $\mathsf{p1}$ and $\mathsf{p2}$ at addresses ι_1 and ι_2, sharing their x and y coordinates:

$$\sigma_1(\iota_1) = [\![\mathsf{d_1} = \iota_2 \quad \| \quad \mathsf{x} = 5]\!]$$
$$\sigma_1(\iota_2) = [\![\mathsf{d_1} = \iota_1 \quad \| \quad \mathsf{y} = 5]\!]$$

In δ the update $\iota_1.\mathsf{y} \blacktriangleleft 2$ will update y in the object at ι_2 and similarly $\iota_2.\mathsf{x} \blacktriangleleft 4$ will update the object at ι_1. For:

$$\sigma_2 = \sigma_1\{\iota_1.\mathsf{x} \lhd^+ 4\}\{\iota_2.\mathsf{y} \lhd^+ 2\}$$

We have:

$$\sigma_2(\iota_1) = [\![\mathsf{d_1} = \iota_2 \quad \| \quad \mathsf{x} = 4]\!]$$
$$\sigma_2(\iota_2) = [\![\mathsf{d_1} = \iota_1 \quad \| \quad \mathsf{y} = 2]\!]$$

On the other hand, $\iota_1.\mathsf{z} \blacktriangleleft 2$ would add method z to the object at ι_1.

Note that the δ method call (*Select*) models "conventional" field access, *and* method call, *and* delegation. For example, for an expression $\mathsf{a.m}$ where a evaluates to an object ι, if ι contains a method m, whose method body is an address, then we have field access. If the ι contains a method m, whose method body is not an address, then we have method call. If however, ι does not contain a method m, but one if its delegates, say ι'' does, then the method from ι'' will be executed. Note however, that the method will be executed in a context where self maps to ι, ie the original receiver of the method call. Therefore, we have delegation, and *not* consultation.

The fact that delegates are names in δ makes it possible to invoke methods directly on specific delegates, using delegate invocation (*Delegate Select*). The delegate invocation (*Delegate Select*) models selection at a particular delegate. Namely, for $\mathsf{a@d.m}$ where a evaluates to an object ι, first its delegate d is searched in ι and its delegates, giving, say, ι''. Then the method body for m is searched in

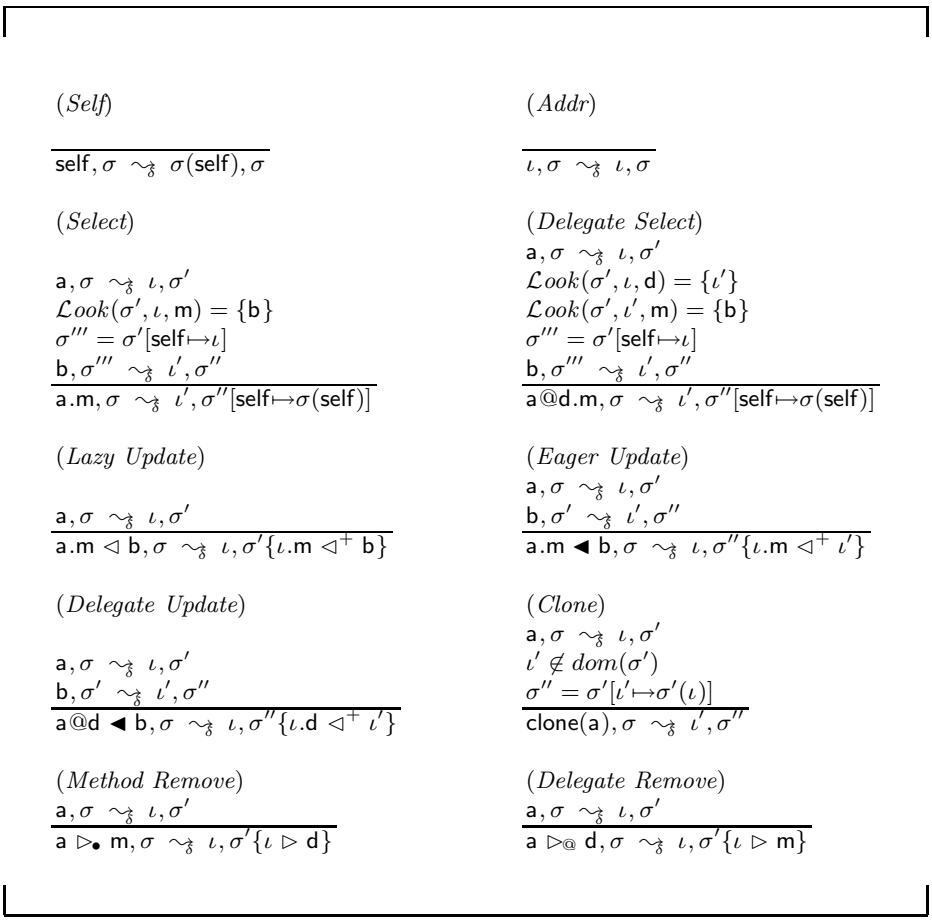

Fig. 7. Operational Semantics of δ

ι'' and its delegates. If such a method is found, then it is executed in a context where the receiver is ι, i.e. the original receiver.

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

Delegate removal (*Delegate Remove*) and method removal (*Method Remove*) first evaluate the receiver (a) then return the receiver with the delegate or method named m removed.

5 δ^- and the Abadi-Cardelli ς-calculus

We compare δ^- with the first order imperative ς-calculus which we shall call ς. δ^- is similar to ς in the following respects:

- They have similar syntax allowing method selection, method update (lazy update in δ^-) and cloning.
- They both use a global state.
- They both represent fields through methods, and field update through method update.

δ differs from ς in that:

- The state is simpler in δ^- than in ς (Objects are directly reflected in δ^-'s store, but in ς objects are reflected through a combination of the store and the stack).
- ς allows "expanded objects" e.g. $[\![\mathsf{m} = \varsigma(\mathrm{x}_i)\mathsf{b}_\mathsf{i} : \mathsf{i} \in 1 \,, ..., \, \mathsf{n} \,]\!]$, in the syntax, whereas δ^- expects the objects to be in the store, and allows addresses in the syntax.
- No stacks are required in δ^-. In ς stacks associate variables with objects.
- δ^- uses self where ς uses $\varsigma(\mathrm{x})$ bindings. Thus, in ς through the use of several $\varsigma(\mathrm{x})$ one can have many bindings; this however can be reflected in δ^- through the use of the object's address.
- δ^- directly reflects delegation rather than encode it, and thus, has the expected behaviour in the case where a method is modified in the delegate after setting the object's delegate.
- δ^- allows extension for objects by new methods, while ς only allows update of existing methods.
- δ^- uses large step substitution based operational semantics, whereas ς uses a large step closure based operational semantics, which could be considered to be more implementational.

From the ς to δ^- We have formulated mappings from δ^- to ς and from ς to δ^-. This required the construction of a non-trivial correspondence between the ς state and the δ^- state. To distinguish δ^- entities from ς entities we use superscripts $^{\delta}$ and ^{AC} respectively. Rewrites in ς are denoted by ... $\leadsto_{AC}$ ... whereas rewrites in δ are denoted by ... $\leadsto_{\delta}$ ...

Furthermore, in [2] we defined relations $\preceq_r$ and $\preceq_p$, where $\preceq_r$ relates ς expressions, stores and stacks to δ^- expressions and stores, and $\preceq_p$ relates ς object results and stores to δ^- expressions and stores. We express the correspondence between the two calculi:

Theorem 1.

$$If \quad \sigma^{AC} \cdot S^{AC} \vdash \mathsf{e}^{AC} \rightsquigarrow_{AC} \mathsf{v}^{AC}, \sigma_1^{AC} \quad and \quad \mathsf{e}^{AC}, \sigma^{AC}, S^{AC} \preceq_r \mathsf{e}^{\delta}, \sigma^{\delta},$$

$$then \quad \exists\, \iota^{\delta}, \sigma_1^{\delta} : \quad \mathsf{e}^{\delta}, \sigma^{\delta} \rightsquigarrow_{\delta} \iota^{\delta}, \sigma_1^{\delta} \quad and \quad \mathsf{v}^{AC}, \sigma_1^{AC} \preceq_p \iota^{\delta}, \sigma_1^{\delta}$$

Proof by structural induction over the derivation $\sigma^{AC} \cdot S^{AC} \vdash \mathsf{e}^{AC} \rightsquigarrow_{AC} \mathsf{v}^{AC}, \sigma_1^{AC}$.

We have a similar theorem in the other direction, from δ^- to ς, cf [2].

6 Conclusions and Future Work

We have argued that delegation in object based programming supports flexible programs that can adapt to unanticipated change. New behaviour can be encoded in new objects and existing objects can delegate to these new objects. Local maintenance can be done by dynamically updating methods. All these operations can be done long after the system was designed and installed.

We have presented a simple imperative object based calculus with explicit delegation. By representing objects directly in the store and using a substitution based operational semantics we obtained an intuitive, and simple calculus. The calculus is succinct, and as shown in 4, one feature models field access *and* method call *and* delegation. We also argue that by representing delegation explicitly, δ gives the expected behaviour in cases of method update after delegate setting, whereas [1, 3] do not.

It is still an open point whether for practical purposes delegation should be incorporated into class based or into object based languages [17]. In how far the class or the object should be the unit of organisation and reuse is still debatable, for example recently [14] suggest delegation layers and replace multi-class mixin-inheritance by multi-object delegation.

The merit of δ, in our view is that it is succinct, that is correctly reflects the behaviour of delegation, and therefore, we believe it can serve as a formal foundation for both object based and class based languages with delegation (as [1] did for both object and for class based languages).

We would like to develop mappings from simple class based programming languages to δ and from δ to class based languages. We would like to work on a mapping from δ to δ^-. We are also developing a statically typed class based language with delegation.

7 Acknowledgements

We would like to thank Elena Zucca for very detailed comments and useful suggestions, and Mark Skipper for his help and insight. We would also like to thank

colleagues at Imperial College Department of Computing. We are grateful to the anonymous reviews from the USE'02 program committee, for many comments and suggestions, and for the opportunity to see our work through their view point.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, NY, 1996.
2. Christopher Anderson. Delta - An Imperative Object Based Calculus with Delegation - Full Technical report - `http://www.binarylord.com/work`.
3. Viviana Bono and Kathleen Fisher. An imperative first-order calculus with object extension. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1998.
4. P. Costanza. Lava Design and Compiler - Language Design and Compiler - Diploma thesis, Computer Science Department III, University of Bonn. January 1998.
5. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *ECOOP'01*, LNCS 2072, pages 130–149. Springer, 2001.
6. Ole Agesen et al. The SELF 4.0 Programmer's Reference Manual - `http://research.sun.com/self/`, 1995.
7. K. Fisher and J. Mitchell. A delegation-based object calculus with subtyping. In *Fundamentals of Computation Theory (FCT'95)*. Springer LNCS, 1995.
8. Javascript Language Reference. `http://developer.netscape.com/docs/manuals/js/client/jsguide/ClientGuideJS13.pdf`.
9. Günter Kniesel. Delegation for Java – API or Language Extension? Technical report IAI-TR-98-4, ISSN 0944-8535, CS Dept. III, University of Bonn, Germany, May 1998. `http://javalab.cs.uni-bonn.de/research/darwin/`.
10. Günter Kniesel. Type-safe delegation for run-time component adaptation. In Rachid Guerraoui, editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628, pages 351–366. Springer-Verlag, New York, NY, 1999.
11. Günter Kniesel. *Darwin – Dynamic Object-Based Inheritance with Subtyping*. PhD thesis, CS Dept. III, University of Bonn, Germany, 2000.
12. H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 21, pages 214–223, New York, NY, 1986. ACM Press.
13. H. Lieberman. Concurrent object-oriented programming in Act 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, Cambridge (MA), USA, 1987.
14. Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, Malaga, Spain, 2002.
15. Matthias Schickel. Lava: Konzeptionierung und Implementierung von Delegationsmechanismen in der Java Laufzeitumgebung. Diploma thesis, CS Dept. III, University of Bonn, Germany, December 1997. `http://javalab.cs.uni-bonn.de/research/darwin/`.

16. David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Holzle. Organizing programs without classes. *Lisp and Symbolic Computation*, 4(3):0–, 1991.
17. USE'02. Review of our paper.
18. John Viega, Bill Tutt, and Reimer Behrends. Automated delegation is a viable alternative to multiple inheritance in class based languages. Technical Report CS-98-03, 2, 1998.