Delegation-based Semantics for Modularizing Crosscutting Concerns

Hans Schippers * Dirk Janssens
Formal Techniques in Software Engineering
University of Antwerp, Belgium
{hans.schippers,dirk.janssens}@ua.ac.be

Michael Haupt Robert Hirschfeld
Software Architecture Group
Hasso-Plattner-Institut
University of Potsdam, Germany
{michael.haupt,hirschfeld}@hpi.uni-potsdam.de

Abstract

We describe semantic mappings of four high-level programming languages to our delegation-based machine model for aspect-oriented programming. One of the languages is a class-based object-oriented one. The other three represent extensions thereof that support various approaches to modularizing crosscutting concerns. We explain informally that an operational semantics expressed in terms of the model's concepts preserves the behavior of a program written in one of the high-level languages. We hence argue our model to be semantically sound in that sense, as well as sufficiently expressive in order to correctly support features such as class-based object-oriented programming, the open-classes and pointcut-and-advice flavors of aspect-oriented programming, and dynamic layers. For the latter, being a core feature of context-oriented programming, we also provide a formal semantics.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]:
Semantics of Programming Languages—Operational Semantics

General Terms Theory, Languages

Keywords Semantic Mappings, Aspect-oriented Semantics, Context-oriented Programming, Modularization

1. Introduction

In previous work [12], we have introduced a machine model for aspect-oriented programming (AOP), which is centered on *delegation* and relies on the notion of join points as *loci of late binding*, upon whose occurrence *dispatch operations* determine the functionality to execute. The model was shown to elegantly support a wide range of core mechanisms and features of both the object- and aspect-oriented programming paradigms, such as classes, inheritance, open classes [17] in the form of dynamic introductions and pointcut-and-advice based AOP [15]. However, the model's ability to serve as the basis for complete aspect-oriented programming language implementations was not shown.

This paper demonstrates how the operational semantics of *j, ij, aj* [21] and *cj,* four high-level programming languages, can be mapped onto the operational semantics of our machine model, as well as provides an informal argument suggesting their behavior is preserved in the process. Hence, whereas our previous work started from the model itself to demonstrate its capabilities, the model now serves as a target platform for several *existing* languages and the modularization techniques they support. In other words, the model is shown to be sufficiently expressive to meet the requirements of *j*, *ij*, *aj* and *cj*, which support different approaches to modularization.

j is, essentially, a Java subset and hence supports basic object-oriented principles. The other languages are extensions, adding different modularity mechanisms for the implementation of crosscutting concerns. More specifically, *ij* adds inter-type declarations, while *aj* adds aspects, pointcuts and advice. Thus, *ij* represents an implementation of the *open classes* flavor of AOP, and *aj*, one of the *pointcuts-and-advice* flavor [15]. The languages are formally described by means of an operational semantics for each of them [21]. Although *ij*'s semantics is described as a transformation into an equivalent *j* program, we will show our model to be capable of directly supporting *ij*'s features.

cj is our own contribution to the *j* language family. It does not include the features of *ij* and *aj*, but instead adopts context-oriented programming (COP) [5, 13], a layer-based approach to the modularization of crosscutting concerns. Layers allow context-specific behavioral variations to be

* Ph.D. fellowship of the Research Foundation - Flanders (FWO)

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

composed based on the execution context. *cj* can be considered a subset of ContextJ [6].

The contributions of this paper are as follows:

- we show how the semantics of the *j*, *ij*, and *aj* languages [21] can be *directly expressed* in terms of our model's semantics [12],
- we explain informally that these semantics are equivalent with their original counterparts,
- we provide, for the first time, a formal semantics for *cj*, and hence, for one flavor of context-oriented programming, and map it to our model's semantics like for the other three languages.

All of the aforementioned languages have been implemented on top of a prototype implementation of the delegation-based model, and said implementations adhere to the semantics we present. A description of the implementations is out of the scope of this paper.

This paper is organized as follows. The next section briefly reviews our machine model along with its formal semantics. Sec. 3 describes the semantics mappings for all four languages, and additionally introduces a formal semantics for *cj*. Sec. 4 attends to related work. The paper is summarized and future work is discussed in Sec. 5.

2. A Machine Model for Aspect-Oriented Programming

We restrict the presentation of the machine model we use in this work [12] to a brief summary of the model and its semantics to facilitate an easier understanding of the semantic mappings described in Sec. 3. The following sections 2.1 and 2.2 recapitulate, in condensed form, material that has been published in [12], while Sec. 2.3 describes some modifications to the original semantics that are required in the scope of this work.

2.1 The Model

Core features of the model pertain to the representation of application entities and that of join points. The latter are consistently regarded as *loci of late binding*, and hence of virtual functionality dispatch, where dispatch is organized along multiple dimensions. Each dimension is one possible way to choose a particular binding of a piece of functionality to a join point, e. g., the current object, the target of a method call, the invoked method, the current thread, etc.

Objects are, using a prototype-based object-oriented environment, consistently represented as "seas of fragments" [18]: each object is visible to others only in the form of a *proxy*. Messages sent to an object are received by its proxy and *delegated* to the actual object, as displayed in Fig. 1. Classes are represented likewise: each class is a pair of a proxy and an object representing the actual class. Each object references its class by delegating to the class' proxy.

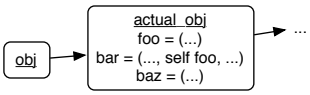

Figure 1. An object is represented as a combination of a proxy and the actual object.

The granularity of the supported join point model is that of message receptions. It is important to note that this granularity exists only at the level of the execution model, where member field access is also mapped to messages. Language implementations on top of the model will map their own join point model to the one defined by the machine model.

A join point's nature as a locus of late binding is realized by means of inserting additional proxy objects in between the proxy and the actual object, or in between the class object's proxy and the actual class-representing object. That way, a message passed on along the delegation chain can be interpreted differently by various proxies understanding it, establishing late binding of said message to functionality. *Weaving*—both static and dynamic—is realized by allowing for the insertion and removal of proxy objects into and from delegation chains.

2.2 Model Semantics

The formal, operational semantics of the machine model, provided in [12], is based on the δ calculus [2]. More specifically, reduction rules are used in order to define an operational semantics function which rewrites a combination of an *expression* and a *store* into an *object address*, representing the result value, and a potentially modified store:

$$\rightsquigarrow_\delta : Exp \times Store \rightarrow Address \times Store$$

At the heart of the semantics are the *Clone* (1) and *Select* (2) operations [12], which respectively handle object creation and message sending:

$$\frac{\begin{array}{c} a, \sigma \rightsquigarrow_\delta \iota, \sigma' \\ \iota' \notin dom(\sigma') \\ \sigma'' = \sigma'[\iota' \mapsto \sigma'(\iota); Del_{\iota'} \mapsto Del_\iota] \\ \iota'' \notin dom(\sigma'') \\ \sigma''' = \sigma''[\iota'' \mapsto [\![]\!]][Del_{\iota''} \mapsto \iota'] \end{array}}{clone(a), \sigma \rightsquigarrow_\delta \iota'', \sigma'''} \quad (1)$$

$$\frac{\begin{array}{c} a, \sigma \rightsquigarrow_\delta \iota, \sigma' \\ Look(\sigma', \iota, m) = (b, \iota_d) \\ \sigma'' = \sigma'[this \mapsto \iota][msg \mapsto m][cur \mapsto \iota_d] \\ b, \sigma'' \rightsquigarrow_\delta \iota', \sigma''' \\ \sigma'''' = \sigma'''[this \mapsto \sigma(this)] \\ \qquad [msg \mapsto \sigma(msg)][cur \mapsto \sigma(cur)] \end{array}}{a.m, \sigma \rightsquigarrow_\delta \iota', \sigma''''} \quad (2)$$

where σ denotes the store, which essentially maps addresses to objects.

An object is represented as a list $[\![m_1 : e_1, \ldots, m_n : e_n]\!]$ of messages it understands, along with their implementations. The store additionally contains, for each address ι, the Del_ι function, which determines the address of the *delegate* of the object at ι. The delegate is the object to which messages are delegated if they are not understood. In general, an object's delegate may depend, for example, upon the currently active thread. However, in the context of this paper, Del_ι will always be a constant function.

Updates to the store are expressed in square brackets, where the $\mapsto$ symbol is used to either assign a new object to an address, or change the value of a particular Del_ι function.

For example, the store $\sigma'[\iota' \mapsto \sigma(\iota)][Del_{\iota'} \mapsto Del_\iota]$ is identical to σ', except that address ι' now holds the same value as the one at address ι (meaning a copy of the object at ι has been stored at ι'), and the constant function $Del_{\iota'}$ is now equal to the constant function Del_ι (meaning the object at ι' now has the same delegate as the object at ι).

Basically, *Clone* (1) creates objects as a pair of objects: an empty *proxy* and the *actual object*, with the proxy simply delegating all messages to the actual object. *Select* (2), on the other hand, looks up an implementation b of a message m, sent to an object (expression) a, by means of the *Look* function, which recursively traverses a's delegate objects until it encounters an implementation of m, and thus encapsulates the delegation mechanism inherent to the system. Its definition is shown in Fig. 2.

Next, b is evaluated while the store is extended to map a number of symbols, representing special variables, to an appropriate value: *this* holds the address of the message receiver object, *msg* holds the message name, and *cur* contains the address of the specific delegate object where the message implementation was eventually found. This information is necessary in order to support *resending* messages further along the delegation chain:

$$\frac{\begin{array}{c} Look(\sigma, Del_{\sigma(cur)}, \sigma(msg)) = (b, \iota_d) \\ \sigma' = \sigma[cur \mapsto \iota_d] \\ b, \sigma' \rightsquigarrow_\delta \iota, \sigma'' \\ \sigma''' = \sigma''[cur \mapsto \sigma(cur)] \end{array}}{resend, \sigma \rightsquigarrow_\delta \iota, \sigma'''} \quad (4)$$

Note that *this* is not updated, and hence remains bound to the original receiver. This matches the common semantics of delegation-based object-oriented programming and is crucial for the proper functioning of the model [12].

2.3 Minor Modifications

In order to facilitate the description of the semantic mappings in Sec. 3, we apply a few superficial modifications, which do not, however, change the actual semantics.

Store vs. Stack/Heap

The store variable σ actually models the combination of a *heap* and a *stack*: Apart from storing all objects, it additionally takes care of stack variables such as *this*. However, this is a non-fundamental design choice [21, p. 66], and heap and stack may just as well be modeled separately by means of two variables h and s. Both approaches are equivalent, provided that, in case of a single store, care is taken to rebind stack variables to their previous value after a method call, as opposed to passing a newly constructed stack frame along with each method call. For example, the *Select* rule might just as well be modeled as follows:

$$\frac{\begin{array}{c} a, h, s \rightsquigarrow_\delta \iota, h' \\ Look(h', \iota, m) = (b, \iota_d) \\ b, h', \{this \mapsto \iota, msg \mapsto m, cur \mapsto \iota_d\} \rightsquigarrow_\delta \iota', h'' \end{array}}{a.m, h, s \rightsquigarrow_\delta \iota', h''} \quad (5)$$

In the following, this style will be adopted for all rules.

Message Parameters

The delegation-based machine model does not model a parameter passing mechanism. However, a strategy supporting exactly one formal parameter which is always called x, similar to the one used in the original definitions of the *j* language family [21], can be straightforwardly implemented. It comes down to simply passing one more variable on the stack (or store). Hence, where convenient, alternative definitions of the *Select* and *Resend* rules, as shown in Fig. 3, will be used.

Note that, for reasons of simplicity, we assume a message to be uniquely identified by its name. Parameter overloading is disregarded. Multiple parameters can be simulated by regarding the formal parameter x as a container object enclosing the actual parameters.

3. Semantic Mappings

This section will introduce the *j, ij, aj* [21] and *cj* languages, along with their operational semantics. For each of these languages, we will express these semantics alternatively using the model semantics from Sec. 2. We will then explain informally that these semantics result in the same behavior, i. e., evaluating a certain expression results in the same value and the same side-effects on the heap. A formal proof for this claim is considered future work.

The original operational semantics of the languages is expressed by means of a rewriting function defined through a number of reduction rules [21]. This strategy is very similar to the one followed in Sec. 2, hence the use of the same symbol $\rightsquigarrow$, albeit without the δ subscript as it is not based on the δ calculus:

$$P \vdash exp, h, s \rightsquigarrow \iota, h'$$

Note that, compared to the rewrite function in Sec. 2, an extra parameter P is added. It represents the source code of the current program, and is used in some reduction rules in order to extract relevant information such as method declarations or inheritance hierarchies.

$$Look(\sigma,\iota,m) = \begin{cases} (b,\iota) & if\ \sigma(\iota) = [\![\ldots m : b \ldots]\!] \\ Look(\sigma, Del_\iota, m) & otherwise \end{cases} \tag{3}$$

Figure 2. *Look* function.

$$\frac{\begin{array}{c} a_1,h,s \rightsquigarrow_\delta \iota,h' \\ a_2,h',s \rightsquigarrow_\delta \iota',h'' \\ Look(h'',\iota,m) = (b,\iota_d) \\ b,h'',\{this \mapsto \iota, x \mapsto \iota', msg \mapsto m, cur \mapsto \iota_d\} \rightsquigarrow_\delta \iota'',h''' \end{array}}{a_1.m(a_2),h,s \rightsquigarrow_\delta \iota'',h'''} \tag{6}$$

$$\frac{\begin{array}{c} a,h,s \rightsquigarrow_\delta \iota,h' \\ Look(h',Del_{s(cur)},s(msg)) = (b,\iota_d) \\ b,h',\{this \mapsto s(this), msg \mapsto s(msg), cur \mapsto \iota_\delta, x \mapsto \iota\} \rightsquigarrow_\delta \iota',h'' \end{array}}{resend(a),h,s \rightsquigarrow_\delta \iota',h''} \tag{7}$$

Figure 3. Alternative definitions of *Select* and *Resend*, supporting a formal parameter x.

3.1 Class-Based Object-Oriented Programming in *j*

This section handles *j*, which is a subset of Java and supports basic object-oriented principles such as object instantiation, inheritance and method invocation.

Syntax

An EBNF-style definition of the syntax of *j* [21] is given in Lst. 1. It is rather minimal, with a `PROGRAM` consisting of a number of `CLASS` elements, which in turn encapsulate `FIELD` and `METHOD` constructs. Method bodies are expressions, where expressions may be recursively concatenated. Methods always accept exactly one parameter x, which can be referred to in a method body, along with the *this* variable. Finally, three special values *true*, *false* and *null* are available, which we will assume to be predefined objects. Hence, the value of an expression will always be an object.

```
1  PROGRAM ::= CLASS*
2  CLASS   ::= class CLSNAME ext CLSNAME { DECL* }
3  DECL    ::= FIELD | METHOD
4  FIELD   ::= TYPE IDENT
5  METHOD  ::= TYPE IDENT ( TYPE x ) { EXP }
6  EXP     ::= SPECIAL
7            | VAR
8            | new CLSNAME
9            | EXP . IDENT
10           | EXP . IDENT := EXP
11           | EXP . IDENT ( EXP )
12           | EXP == EXP
13           | EXP ; EXP
14           | if ( EXP ) { EXP } else { EXP }
15 SPECIAL ::= true | false | null
16 VAR     ::= this | x
```

Listing 1. *j* syntax.

Objects, Classes and Inheritance

In *j*, classes are static descriptions of the structure of their instances. Instances are created by analyzing that description to find out which fields memory should be allocated for. On the heap, an object is represented as a combination of a map of the names of these fields to their values, and the name of the class it belongs to. This is expressed by the new^ℓ operation [21], which essentially boils down to:

$$\begin{array}{c} new^\ell(P,C) = [\![f_1 : \text{null} \ldots f_n : \text{null}]\!]^C \\ \text{where} \\ AllFields(P,C) = f_1 \ldots f_n \end{array} \tag{8}$$

The *AllFields* function recursively finds all fields of a class C, defined in a program P, as well as its super classes. The reference to an object's class (in superscript in that object's representation) is needed to provide an entry point for method lookup. The reduction rule for class instantiation is as follows:

$$\frac{\begin{array}{c} \iota \notin dom(h) \\ new^\ell(P,C) \neq Undefined \end{array}}{P \vdash new\,C,h,s \rightsquigarrow \iota,h[\iota \mapsto new^\ell(P,C)]} \tag{9}$$

In an object-based setting, the concept of a class does not exist, as the only entities available are objects. As the *Clone* operation (cf. Sec. 2, definition (1)) basically copies an existing object, it seems natural to provide a so-called *prototype* object for each class on the heap, which encapsulates messages corresponding to the fields of that class and all of its super classes. This prototype can then be cloned in order to create instances. The implementations of said messages will hold the field values. Additionally, the class prototype should have a delegate with all method implementations of that class, which in turn should have a delegate with all implementations of that class' super class, and so on. This way, instances will understand the corresponding messages via delegation. Finally, in order to be able to support class-wide manipulations of the delegation chain, a proxy object should be inserted for each class, as outlined in [12]. An example of such a configuration is displayed in Fig. 4.

$$
\begin{aligned}
h_P^j = \quad & \{C_1 \mapsto \iota_{proxy,1}, \ldots, C_n \mapsto \iota_{proxy,n}, ClPt(C_1) \mapsto \iota_1, \ldots ClPt(C_n) \mapsto \iota_n, \\
& \iota_1 \mapsto [\![f_{1,1} : \text{null} \ldots f_{1,k_1} : \text{null}]\!], \ldots, \iota_n \mapsto [\![f_{n,1} : \text{null} \ldots f_{n,k_n} : \text{null}]\!], \\
& \iota_{proxy,1} \mapsto [\![ ]\!], \ldots, \iota_{proxy,n} \mapsto [\![ ]\!], \\
& \iota_{meth,1} \mapsto [\![m_{1,1} : b_{1,1}, \ldots, m_{1,p_1} : b_{1,p_1}]\!], \ldots, \iota_{meth,n} \mapsto [\![m_{n,1} : b_{n,1}, \ldots, m_{n,p_n} : b_{n,p_n}]\!], \\
& Del_{\iota_1} = \iota_{proxy,1}, \ldots, Del_{\iota_n} = \iota_{proxy,n}, \\
& Del_{\iota_{proxy,1}} = \iota_{meth,1}, \ldots, Del_{\iota_{proxy,n}} = \iota_{meth,n}, \\
& Del_{\iota_{meth,1}} = h_P^j(super^\ell(P, C_1)), \ldots, Del_{\iota_{meth,n}} = h_P^j(super^\ell(P, C_n))\}
\end{aligned}
\tag{10}
$$

where

$\forall i : C_i \in Classes(P), AllFields(P, C_i) = f_{i,1} \ldots f_{i,k_i}, \forall j : T\ m_{i,j}(T'\ x)\{b_{i,j}\} \in getMethods(C_i)$

$ClPt : \text{CLSNAME} \rightarrow Address$

$\iota_1 \ldots \iota_n, \iota_{proxy,1} \ldots \iota_{proxy,n}, \iota_{meth,1} \ldots \iota_{meth,n}$ are unique

Figure 5. Prepared heap for *j* programs.

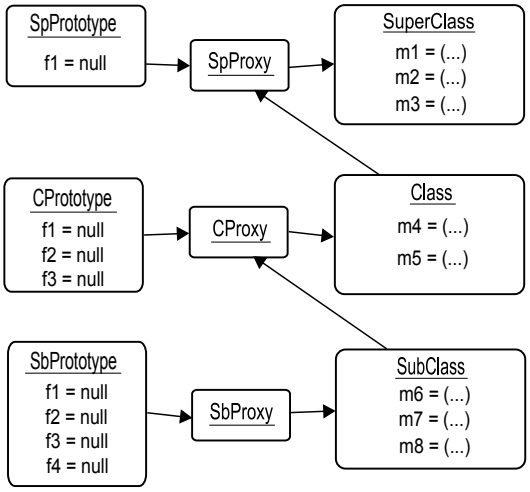

Figure 4. Example of a prepared heap

In order to model this semantically, we assume that, in an object-based setting, a *j* program always executes in the context of a *prepared heap*, rather than an empty one, in a similar fashion as is done for aspects in the *aj* language [21, p. 120]. The heap is assumed to store, beside objects and the *Del* function, mappings of class names to addresses, in order to ensure that classes can be looked up by name, as well as the *ClPt* function, which associates each class with its class prototype. The definition of the prepared heap is displayed in Fig. 5.

Recall that the Del_ι function determines the delegate of the object at address ι, while *AllFields* returns the fields of a class and all its super classes, and *getMethods* [21] results in a collection of the methods defined by one specific class. Finally, the *super* function [21] determines the name of a class' super class. If care is taken to always install a class' super class on the heap prior to the class itself, the super class' proxy can be looked up by means of this name, which is then installed as the delegate of the class' method object. For completeness, a predefined *Object* class could be

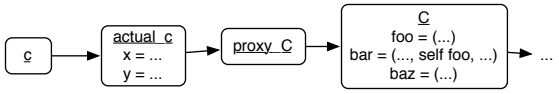

Figure 6. Result of class instantiation with delegation-based semantics.

installed on the heap, serving as the super class of the first user-defined class.

With this prepared heap in place at the start of program execution, instance creation in the machine model works as follows:

$$
\frac{P \vdash clone(ClPt(C)), h, s \rightsquigarrow_\delta \iota, h'}{P \vdash new\ C, h, s \rightsquigarrow_\delta \iota, h'} \tag{11}
$$

The *new* operation will thus result in a proxy (created by *clone*), delegating all messages to a copy of *C*'s class prototype, the latter being available on the heap at $ClPt(C)$. As the $Del_{ClPt(C)}$ function is also stored on the heap, and hence copied as well, the copy ends up delegating to *C*'s proxy, and hence eventually to the object holding *C*'s methods. This is displayed in Fig. 6.

Method Lookup

Although it is now established how classes and objects are dealt with in an object-based setting, it should still be verified that such strategy indeed results in the same program behavior. More specifically, a method call (or message send) should, both in original *j* semantics as well as in delegation-based semantics, result in the same method (or message) implementation being found and executed.

The two relevant reduction rules are the *Select* rule (cf. Sec. 2, definition (6)) on the one hand, and the *j* rule for method calls on the other [21]:

$$
\frac{\begin{array}{c} P \vdash e_0, h, s \rightsquigarrow \iota, h' \\ P \vdash e_1, h', s \rightsquigarrow \iota', h'' \\ M^\ell(P, type(h'', \iota), m) = T\ m(T'\ x)\{e\} \\ P \vdash e, h'', \{this \mapsto \iota, x \mapsto \iota'\} \rightsquigarrow \iota'', h''' \end{array}}{P \vdash e_0.m(e_1), h, s \rightsquigarrow \iota'', h'''}
$$

$$M^\ell : \text{PROGRAM} \times \text{CLSNAME} \times Identifier \rightarrow \text{METHOD}$$

$$M^\ell(P,C,m) = \begin{cases} Undefined & if\ C = \text{Object} \\ M & if\ getMethods(C)/_{\{m\}} = M \\ M^\ell(P, super(P,C), m) & if\ getMethods(C)/_{\{m\}} = \varepsilon \\ Undefined & otherwise \end{cases} \tag{12}$$

Figure 7. M^ℓ function.

where the *type* function [21] determines the dynamic type of an object. The *msg* and *cur* variables in definition (6) can be ignored since they are only ever used while *resending* messages, which never occurs here. Apart from that, the only difference between both rules is in the M^ℓ and *Look* (cf. Sec. 2, definition (3)) functions, which capture the respective lookup algorithms. Hence, the crucial question is now whether they traverse the space of method implementations in the same way. The M^ℓ function is defined as in Fig. 7 [21].

The *getMethods* helper function [21] in the definition results in a collection of the methods defined by one specific class and the *super* function [21] determines the name of a class' super class, while $X/_{\{q\}}$ [21, p. 50] is the domain restriction function, which basically restricts a set of syntactic constructs X to the ones containing an identifier q. Note that words in smallcaps, such as METHOD, represent their corresponding syntactic constructs as defined in Lst. 1.

M^ℓ first checks whether a method named m is defined by the class passed as a parameter, which is the class to which the target object of the method call belongs. If not found there, the chain of super classes will be traversed recursively until m is encountered. Note that the checks are performed based on the program text.

The *Look* function, on the other hand, actually starts its search in the target object (rather, on its proxy), and recursively traverses that object's delegation chain until an object is encountered which has an implementation for m. Considering a proxy by definition does not respond to any messages, and considering the setup of the prepared heap (10), the first candidate object is the actual object, i. e., the copy of the class prototype which was created during instantiation. This corresponds to *actual_c* in Fig. 6. However, as this object, by definition, contains only fields, the search continues, passing the class proxy and ending up in the object holding the method implementations of the target object's class. If a suitable method implementation is found here, it is returned; if not, the search continues, passing the class proxy of the super class, and ending up in the object holding the method implementations of the super class, and so on.

In conclusion, the lookup algorithm does indeed encounter all candidate method implementations in the same order, and hence yields the same result. It is important to realize that *the heart of the semantic mapping is in fact in prepared heap construction*. It is the specific configuration of this heap which causes j method calling and message sending to deliver the same results.

Field Access

Although it has now been argued that method calls will result in the same behavior in both versions of operational semantics, we still need to show that heap access semantics are preserved. While object creation, which has been covered above, is obviously part of that, there is no assurance so far that field access works correctly. The j reduction rule for getting a field value is as follows:

$$\frac{P \vdash e,h,s \rightsquigarrow \iota,h' \quad h'(\iota)(f) = \iota'}{P \vdash e.f,h,s \rightsquigarrow \iota',h'} \tag{13}$$

This means that heap content is simply checked at the target object's address, which, considering the definition of new^ℓ above (8), indeed contains values for all its fields.

In the object-based machine model, fields, just like methods, are mapped to messages, where the message implementations are simply the field values. Hence, the *Select* rule (cf. Sec. 2 (5)) should be applied. By construction of the prepared heap (10), we know that the receiver of the message is a proxy, hence the message implementation will be found in the actual object. By the definition of *Look* (cf. Sec. 2 (3)), this is indeed at $h'(\iota)(f)$. As we know the implementation to already be an address, the last line of *Select*, evaluating the implementation, can be ignored. Hence, the remainder is equivalent to (13).

Field updates, which j handles in a straightforward way by modifying the appropriate location on the heap, should obviously be handled by replacing the corresponding message implementation.

Other *j* Operations

j supports other operations besides method calls and field access, such as testing for equality and an *if-then-else* construct. However, as the machine model only supports objects and messages, all these should be implemented as message sends, in the spirit of purely object-oriented languages such as Smalltalk. In such a scenario, their semantics are covered by the previous discussion on method calls.

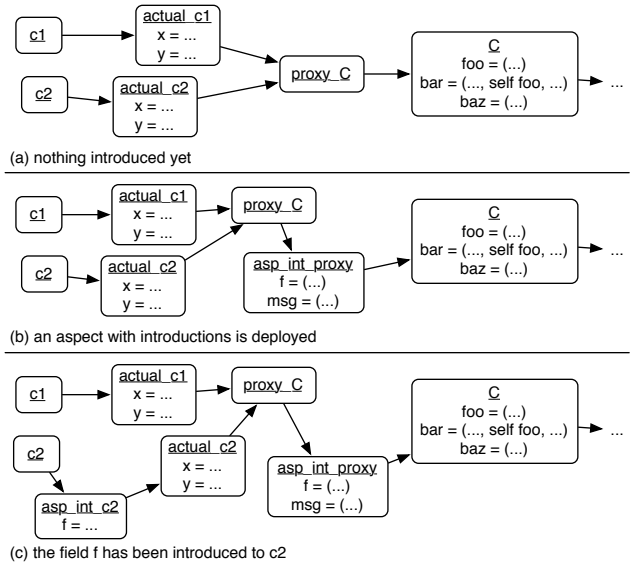

Figure 8. Dynamic introduction of a field `f` and a message `msg`.

3.2 Inter-Type Declarations in *ij*

ij [21] is an extension of *j* which adds inter-type declarations. More specifically, it allows classes to declare fields and methods which actually belong to another class. To this end, it provides a simple extension to the *j* syntax, as listed in Lst. 2.

```
1 CLASS   ::= class CLSNAME ext CLSNAME { DECL* ITD* }
2 ITD     ::= CLSNAME <-- DECL
```

Listing 2. *ij* syntax.

Semantics for this construct are defined in [21] in terms of a rather straightforward weaving approach: Initially, a syntax transformation is applied, turning each `ITD` into a `DECL` in the proper class. This results in a valid *j* program, the semantics of which are known.

In our machine model, however, inter-type declarations are supported directly, and in fact implemented as dynamic introductions [12]. Each inter-type method declaration results in a new proxy object being created and inserted in the delegation chain of the target class. For field introductions, a class-wide method is added as well, execution of which results in another proxy, this time instance-local, to be dynamically inserted in the delegation chain of the instance object which received the message. All this is displayed in Fig. 8, where in (b) actually just one class-wide proxy is inserted for two introductions. While this is indeed a possible optimization, we will assume here, for the sake of simplicity, that each inter-type declaration results in a new proxy.

Method Introductions

We now need to verify that, given an inter-type declaration `C <-- T m(T' x) { b }`, inserting a proxy with an implementation for *m* results in the same behavior as if *m* would have been declared in *C*'s class definition. Formally, given a prepared heap h_P such as (10) in Sec. 3.1, the effects of this inter-type declaration are the following:

$$h'_P = h_P[\iota_{ip} \mapsto [\![m : b]\!]][Del_{\iota_{ip}} \mapsto Del_{h_P(C)}][Del_{h_P(C)} \mapsto \iota_{ip}]$$

where ι_{ip} is an unused address in h_P. In other words, *C*'s class proxy now delegates to the new proxy holding an implementation for *m*, which in turn delegates to *C*'s original method object.

Recalling the *Look* function (cf. Sec. 2 (3)), which is at the heart of method call semantics in the delegation-based model, it now becomes clear that this semantics is indeed satisfactory: As *Look* recursively traverses the delegation chain through the *Del* function, it will eventually encounter *m*'s implementation when *m* is called on an instance of *C*. Moreover, as the new proxy delegates to *C*'s original method object, implementations of other methods defined by *C* will still be found as well.

Note that these heap modifications do not take place at runtime. Rather, they cause the prepared heap to have a slightly different configuration at program start, mimicking the static character of inter-type declarations in *ij*. However, our machine model allows for similar delegation chain modifications during program execution as well, which occurs for example in the context of field introductions.

Field Introductions

As hinted above, field introductions require an additional step compared to method introductions. Consider an inter-type field declaration `C <-- T f`. Its effects on a prepared heap are similar at the first glance:

$$h'_P = h_P[\iota_{ip} \mapsto [\![f : b_{ip}]\!]][Del_{\iota_{ip}} \mapsto Del_{h_P(C)}][Del_{h_P(C)} \mapsto \iota_{ip}]$$

Note, however, that b_{ip}, the implementation of the message f, is this time not provided as part of the syntactic program. Instead, it has specific semantics, resulting in the installation of another proxy in the delegation chain of the target object, which is available on the stack through *this*. Formally, the semantics of its execution is as follows:

$$\frac{\begin{array}{c}\iota_{ip'} \notin dom(h) \\ h' = h[\iota_{ip'} \mapsto [\![f : \text{null}]\!]] \\ \iota = s(this) \\ h'' = h'[Del_{\iota_{ip'}} \mapsto Del_\iota][Del_\iota \mapsto \iota_{ip'}] \\ \iota.f, h'', s \rightsquigarrow_\delta \iota', h'''\end{array}}{b_{\iota p}, h, s \rightsquigarrow_\delta \iota', h'''}$$

where we actually know that $\iota' = null$ and $h''' = h''$, because f being sent again will now encounter an implementation at $\iota_{ip'}$, which is *null*, and hence trivially evaluates to itself without modifying the heap.

The important observation, however, is that the method implementation which is installed at ι_{ip} is only executed

at first field access, and eventually results in *null* being returned, exactly as if the field had been declared as part of *C*'s class definition, which would have resulted in it being part of the prepared heap (cf. Sec. 3.1 (10)) and initialized to *null* there. Semantics of *new* (cf. Sec. 3.1 (11)) would then have copied it during cloning to become part of the instance, where it would have been available immediately at first access.

For all subsequent accesses, a similar argument as the one for method introductions applies.

3.3 Before and After Advice in *aj*

aj [21] is an extension of *j* which adds support for *before* and *after* advice at *call*, *get* and *set* join points. Since in *aj* a call join point always coincides with an execution join point, *call* actually represents both. Syntactically, *aj* adds the `ASPECT`, `ADVICE` and `POINTCUT` constructs compared to *j*, as displayed in Lst. 3.

```
1 PROGRAM ::= < CLASS | ASPECT >*
2 ASPECT  ::= aspect CLSNAME { DECL* ADVICE* }
3 ADVICE  ::= < before | after > POINTCUT { EXP }
4 POINTCUT::= call ( TYPE TYPE.IDENT(TYPE) )
5           | get ( TYPE TYPE.IDENT )
6           | set ( TYPE TYPE.IDENT )
7 VAR     ::= this | x | s | r | v
```

Listing 3. *aj* syntax.

Additionally, it introduces the stack variables *s* for the caller object, *r* for the target object and *v* for the new value of a field update. These variables must only occur in an advice body. Note that the original *aj* supports a limited form of wildcards in pointcut specifications, but that feature is left out for simplicity reasons.

Semantics are expressed in terms of a number of syntax analyzing helper functions, which determine whether some advice is applicable at field get, set or method call, and execute it accordingly.

Our machine model, however, supports advice by inserting a proxy in the delegation chain of the appropriate class, as displayed in Fig. 9. Actually, this technique is sufficiently powerful in order to support *around* advice with *proceed* as well. Although this is not supported by *aj*, *proceed* is covered later on in Sec. 3.4 as part of *cj*, with essentially identical semantics as would be required here.

As the model currently does not keep track of type information for fields and methods, we will disregard the type specifiers in a `POINTCUT` and hence assume that the method or field name and its owning class uniquely determine a method or field.

get Join Points

Semantics for getting a field's value in *aj* are essentially given by the reduction rule shown in Fig. 10 [21]. First, the F_t^ℓ function is applied, which is similar to M^ℓ (cf. Sec. 3.1 (12)), except that it looks up a *field*'s declaration in a class hierarchy, and returns the name of the class where it was declared along with it, as shown in Fig. 11. Recall that the *type* function [21] determines the dynamic type of an object.

Next, the $advices^\ell$ function computes a number of ordered sets of advice applicable at the join point in question. The fact that there are multiple orderings to choose from is due to there being no way to express dominance of one aspect over another. Hence the only requirement is that multiple advice applying to the same join point and *defined within the same aspect* should be executed in order of syntactical appearance.

Each advice is uniquely identified by a symbol a_i which is used by the *owningAspect* function to determine which aspect that advice belongs to. This is important since the *this* variable should, during advice execution, be bound to a singleton object which has memory for this aspect's fields. This implies a prepared heap, which is constructed as follows:

$$h_P^{aj} = \{C_1 \mapsto \iota_1, \ldots, C_n \mapsto \iota_n, \iota_1 \mapsto new^\ell(P,C_1), \ldots, \iota_n \mapsto new^\ell(P,C_n)\} \quad (15)$$

where
$\forall i : C_i \in Aspects(P)$
$\iota_1 \ldots \iota_n$ are unique

Note that this prepared heap is still part of the *aj* semantics as defined in [21], and the fact that new^ℓ (cf. Sec. 3.1 (8)) can be applied to an `ASPECT` is due to the syntactical similarity of `ASPECT` and `CLASS`.

When the set of applicable advice has been determined, all *before* advice is executed. During advice execution, besides *this*, the *s* and *r* variables are made available on the stack, and bound to the caller and target objects, respectively.

After all *before* advice has been executed, the actual field value is fetched and finally all *after* advice is executed. Note that it is the field value ι' which is indeed the result value of the complete expression.

In our machine model, in order to obtain equivalent semantics, a somewhat more extensive prepared heap is needed, which starts out with a setup holding the contents of (10) provided in Sec. 3.1 in order to support classes. In addition, it requires a singleton object for each aspect, but since there is no reason for representing these as a combination of a proxy and an actual object, *aj* semantics can be retained. If we assume $\iota_1 \ldots \iota_n$ to be unused in the prepared heap of (10) in Sec. 3.1, and aspects to have globally unique names, the required prepared heap h_P, so far, is intuitively the *union* of (10) and (15).

However, this is not yet sufficient. In order to allow class-wide advice for fields, we have to deal with the fact that field messages are actually understood in instances, and hence never reach proxies installed in the delegation chain of the class. This problem can be solved by installing getter and setter messages in the class' method object, and ensuring that field access is always performed via these accessor messages. Said messages can easily access the field value by

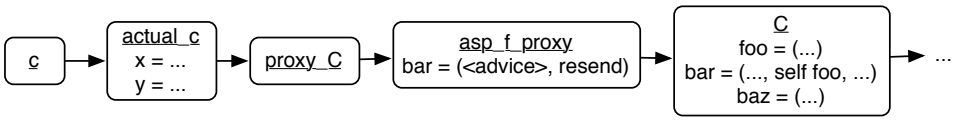

Figure 9. Class-wide advice for *bar* applied.

$$\frac{\begin{array}{c} P \vdash e,h,s \rightsquigarrow \iota,h_1 \\ F_t^\ell(P,type(h_1,\iota),f) = (T\ f,C) \\ A_1 \ldots A_k \in advices^\ell(P,before,get(C.f)) \\ A_{k+1} \ldots A_n \in advices^\ell(P,after,get(C.f)) \\ \forall i : A_i = a_i : \alpha\ get(C.f)\{e_i\},\ owningAspect^\ell(P,a_i) = O_i \\ \alpha \in \{before,after\} \\ P \vdash e_1,h_1,\{\mathrm{s} \mapsto s(this),\mathrm{r} \mapsto \iota,this \mapsto h_1(O_1)\} \rightsquigarrow \iota_1,h_2 \\ \ldots \\ P \vdash e_k,h_k,\{\mathrm{s} \mapsto s(this),\mathrm{r} \mapsto \iota,this \mapsto h_k(O_k)\} \rightsquigarrow \iota_k,h_{k+1} \\ h_{k+1}(\iota)(f) = \iota' \\ P \vdash e_{k+1},h_{k+1},\{\mathrm{s} \mapsto s(this),\mathrm{r} \mapsto \iota,this \mapsto h_{k+1}(O_{k+1})\} \rightsquigarrow \iota_{k+1},h_{k+2} \\ \ldots \\ P \vdash e_n,h_n,\{\mathrm{s} \mapsto s(this),\mathrm{r} \mapsto \iota,this \mapsto h_n(O_n)\} \rightsquigarrow \iota_n,h_{n+1} \end{array}}{P \vdash e.f,h,s \rightsquigarrow \iota',h_{n+1}} \tag{14}$$

Figure 10. Semantics for getting a field value in *aj*.

$$F_t^\ell :: \textsc{program} \times \textsc{clsname} \times Identifier \rightarrow \textsc{field} \times \textsc{clsname}$$

$$F_t^\ell(P,C,f) = \begin{cases} Undefined & if\ C = \mathsf{Object} \\ (F,C) & if\ getFields(P,C)/_{\{f\}} = F \\ F_t^\ell(P,super(P,C),f) & if\ getFields(P,C)/_{\{f\}} = \varepsilon \\ Undefined & otherwise \end{cases}$$

Figure 11. F_t^ℓ function, looking up a field declaration in a class hierarchy, as well as the name of the class where the declaration is found.

sending the actual field message to *this*. For example, the getter method for a field f of a class C is now part of C's method object:

$$\iota_{meth} \mapsto [\![m_1 : b_1, \ldots, m_n : b_n, getF : this.f, \ldots]\!]$$

Note that this does not require *aj* programs to provide such accessor methods, but that the compiler should appropriately translate field access into the corresponding message sends.

Finally, we can describe how advice applying to a join point `get(T C.f)` (where we disregard `T`) affects the prepared heap. This is shown in Fig. 12.

Basically, what happens is that each *before* advice is translated to a proxy which understands the getter message of the field in question. It provides an implementation which, after execution of the actual advice body, resends the message along the delegation chain until it reaches the original getter implementation which returns the field value. Semantics of resending a message are provided by the *Resend* reduction rule (cf. Sec. 2 (4)). *after* advice is treated identically, except for the fact that the resend now occurs prior to execution of the advice body.

Furthermore, the heap also contains a function *Asp*, which keeps track of the aspect each advice was defined in. This is important in order to be able to properly bind *this* during advice execution.

It should now be clear that, similarly to Sec. 3.2, the existing semantics of the *Select* rule (cf. Sec. 2 (5)) and the lookup algorithm incorporated in the *Look* function (cf. Sec. 2 (3)) are essentially sufficient for properly executing all before and after advice as well as the actual field access. However, care should be taken to properly bind the stack variables. Hence *Select* is slightly modified, as shown in Fig. 13.

The *Select* rule is responsible for message sending in general, and is oblivious as to whether it is executing advice or a normal method. This is where the *Asp* function is useful, as, in case of a method belonging to an advice proxy, it

$$
\begin{aligned}
h'_P = \quad & h_P[\iota_{ap,1} \mapsto [\![getF : e_1; resend]\!]][Del_{\iota_{ap,1}} \mapsto Del_{h_P(C)}][Del_{h_P(C)} \mapsto \iota_{ap,1}] \\
& \cdots \\
& [\iota_{ap,k} \mapsto [\![getF : e_k; resend]\!]][Del_{\iota_{ap,k}} \mapsto Del_{h_P(C)}][Del_{h_P(C)} \mapsto \iota_{ap,k}] \\
& [\iota_{ap,k+1} \mapsto [\![getF : resend; e_{k+1}]\!]][Del_{\iota_{ap,k+1}} \mapsto Del_{h_P(C)}][Del_{h_P(C)} \mapsto \iota_{ap,k+1}] \\
& \cdots \\
& [\iota_{ap,n} \mapsto [\![getF : resend; e_n]\!]][Del_{\iota_{ap,n}} \mapsto Del_{h_P(C)}][Del_{h_P(C)} \mapsto \iota_{ap,n}] \\
& [Asp(\iota_{ap,1}) \mapsto O_1] \ldots [Asp(\iota_{ap,n}) \mapsto O_n]
\end{aligned}
\tag{16}
$$

where

$\iota_{ap,1} \ldots \iota_{ap,n}$ are unique *and* $\forall i : \iota_{ap,i} \notin dom(h_P)$
$A_k \ldots A_1 \in advices^{\ell}(P, before, get(C.f))$
$A_{k+1} \ldots A_n \in advices^{\ell}(P, after, get(C.f))$
$\forall i : A_i = a_i : \alpha\ get(C.f)\{e_i\},\ O_i = owningAspect^{\ell}(P, a_i)$
$Asp : Address \rightarrow \text{CLSNAME}$
$\alpha \in \{before, after\}$

Figure 12. Updates to the prepared heap upon applying advice to a join point of the form `get(T C.f)`

$$
\frac{\begin{array}{c} a,h,s \rightsquigarrow_{\delta} \iota,h' \\ Look(h',\iota,m) = (b,\iota_d) \\ s' = \begin{cases} \{\mathrm{s} \mapsto s(this), \mathrm{r} \mapsto \iota, this \mapsto h'(A), msg \mapsto m, cur \mapsto \iota_d\} & if\ Asp(\iota_d) = A \\ \{this \mapsto \iota, msg \mapsto m, cur \mapsto \iota_d\} & if\ Asp(\iota_d) = Undefined \end{cases} \\ P \vdash b,h',s' \rightsquigarrow_{\delta} \iota',h'' \end{array}}{P \vdash a.m,h,s \rightsquigarrow_{\delta} \iota',h''}
$$

Figure 13. Slight modification of *Select*, in order to correctly bind stack variables.

returns the name of the aspect to which that advice belongs, and returns *undefined* otherwise. In case of advice, *this*, *s* and *r* are bound to the owning aspect, sender and receiver objects respectively, whereas in case of a normal method, *this* is bound to the receiver while *s* and *r* are not available.

It is important to realize that the fundamentals of *Select* remain untouched, as the only changes are related to a parameter to the recursive invocation of the rewrite function $\rightsquigarrow_{\delta}$. Modification of the *Look* function, for example, would imply that the delegation mechanism is inadequate, in which case our model would be as well.

Resend semantics need to be adapted in a similar superficial fashion as well, in order to correctly update *this*, and to deal with the case where the next object in the delegation chain which also understands the message is *not* an advice proxy. Indeed, in that situation the special stack variables lose their meaning, and moreover *this* should no longer be bound to an aspect singleton, but to the original receiver, which is stored in *r*. The modified *Resend* rule is shown in Fig. 14.

The only requirement left to verify is advice ordering. As explained before, it should be guaranteed that two advice which are defined within the same aspect are executed in their order of syntactical appearance.

For *before* advice, this means their corresponding proxies should appear in the delegation chain in the same order. For *after* advice, they should appear in reverse order, as the proxy which resends first, will execute its advice last.

As $deploy(proxy, target)$ inserts a proxy in the delegation chain immediately after the target object, this implies the proxy of the *last* before advice should be deployed first and the proxy of the *last* after advice should be deployed last. This is asserted in h'_P (16) by first deploying *before* advice proxies in reverse order ($A_k \ldots A_1$ are the before advice) and then deploying *after* advice proxies in straight order ($A_{k+1} \ldots A_n$ are the after advice).

set Join Points

set join points are handled analogously to *get* join points, except that *setF* should be used instead of *getF*, and hence the *Select* variant with parameter support (cf. Sec. 2 (6)) is relevant here.

call Join Points

In the high-level *aj* language, a *call* join point always corresponds to an *execution* join point. For that reason, only *call* is provided in the language. However, at the implementation level, we have to decide whether to affect the caller object or the receiver object. As our delegation-based machine model supports *execution* join points in a natural way, the choice is simple here. However, in case of languages which do have a need to distinguish between *call* and *execution* join points

$$\frac{\begin{array}{c} Look(h, Del_{s(cur)}, s(msg)) = (b, \iota_d) \\ s' = \begin{cases} \{this \mapsto s(\mathrm{r}), msg \mapsto s(msg), cur \mapsto \iota_d\} & if\ Asp(\iota_d) = Undefined \\ s[this \mapsto h(A)][cur \mapsto \iota_d] & if\ Asp(\iota_d) = A \end{cases} \\ P \vdash b, h, s' \rightsquigarrow_\delta \iota, h' \end{array}}{P \vdash resend, h, s \rightsquigarrow_\delta \iota, h'}$$

Figure 14. Modified *Resend* rule.

at the language level, support for *call* join points should be available as well. This is considered future work.

Support for *execution* join points is analogous to *get* join points, but slightly simpler. More specifically, the complication related to the extra getter methods is no longer required, as methods are stored at class level anyway.

3.4 Context-Oriented Programming in *cj*

cj is our own contribution to the *j* language family. Unlike *ij* and *aj, cj* implements the concepts of context-oriented programming (COP) [5, 13], and is basically a subset of ContextJ [6]. Context-oriented programming helps developers to modularize context-dependent behavior. Behavioral variations, or partial definitions of the underlying programming system, are organized in *layers* where each layer aggregates a context-dependent part of a system's property or concern. Layers can be activated and deactivated at runtime, based on the system's context of use. All context-dependent compositions are scoped such that only dedicated system parts are affected, and only for a specific period of time, such as the dynamic extent of a method execution.

cj is defined as an extension to *j*. The syntactic differences are shown in Lst. 4.

```
PROGRAM ::= < CLASS | LAYER >*
CLASS   ::= class CLSNAME ext CLSNAME { EXTDECL* }
LAYER   ::= layer CLSNAME { DECL* }
EXTDECL ::= DECL | LAYER
METHOD  ::= TYPE CLSNAME.IDENT ( TYPE x ) { EXP }
EXP     ::= ...
          | withlayer ( CLSNAME ) { EXP }
          | withoutlayer ( CLSNAME ) { EXP }
          | proceed ( EXP )
VAR     ::= thisLayer | this | x | s
```

Listing 4. *cj* syntax.

Layers can be partially defined at top level as well as class level. A top level layer definition contains method definitions that directly pertain to certain classes, hence the METHOD rule is slightly modified to include the name of said class. Class-level layers implicitly affect methods of the surrounding class, hence the class name in the METHOD rule is redundant in this case. However, for simplicity and uniformity reasons, it is more convenient to use one unique rule for method definitions. The EXP rule has been extended with *withlayer* and *withoutlayer* expressions for activating and deactivating a given layer, while *proceed* allows for proceeding execution in the next layer or invoking the original method implementation. While *s* denotes the caller object just like in *aj*, an additional *thisLayer* keyword is needed here. In code surrounded by *withlayer*, it can be used to access layer fields, while *this* is used to access fields belonging to the class the code occurs in.

Note that, for the semantics, we can even go as far as to assume a program contains only global layers. This is because a class layer, provided its methods are correctly qualified with the name of the enclosing class, may be moved out of the class while the program retains equivalent meaning.

Similarly, *cj* normally allows for layers to be composed of different fragments. More precisely, it is syntactically possible to have several layer definitions with the same name but containing different methods. Semantically, however, that layer is equivalent to the union of all fragments, which means the feature is just syntactic sugar for the case where each layer is defined only once, as a whole. Therefore, this facet can once more be ignored in semantics specification.

For illustration purposes, a *cj* code sample showing an implementation of the well known *observer* pattern [9] is listed in Lst. 5.

```
class Subject {
  Integer attr

  void Subject.setAttr(Integer x) {
    this.attr := x
  }

  void Subject.changed(Object x) {
    out.println(this.attr)
  }
}

layer Observer {
  bool changed

  void Subject.setAttr(Integer x) {
    thisLayer.changed := x.neq(this.attr);
    proceed(x);
    if(thisLayer.changed) {
      this.changed(null)
    } else {
      null
    }
  }
}

class Main {
  Subject s

  void Main.main (Main x) {
    x.s := new Subject;
    x.s.setAttr(12);
    withlayer(Observer) {
      x.s.setAttr(25)
    }
  }
}
```

Listing 5. *cj* code sample of the observer pattern.

Its execution results in output of the number 25, but not 12. This is because the assignment of 12 happens outside a *withlayer* block, and hence the `setAttr` method in the `Observer` layer is not executed. Upon assignment of 25, however, the `Observer` layer is active, hence the `changed` method is eventually called, resulting in output.

We provide a formal semantics for *cj* as an extension of the semantics of *j* [21].

Syntax Functions

For a layer L of the form $L = layer\ C\ \{D_1 \ldots D_n\}$, we define its *identifier* as follows:

$$id : \textsf{LAYER} \longrightarrow \textsf{CLSNAME}$$

$$id(L) = C$$

while its method definitions can be obtained through *getMethods* [21], which works as well for a class $Z = class\ C\ ext\ C'\ \{D_1 \ldots D_n\}$:

$$getMethods : \textsf{PROGRAM} \times \textsf{CLSNAME} \longrightarrow \textsf{METHOD}^*$$

$$getMethods(P,C) = D_{j_1} \ldots D_{j_k}$$

where
$P/_{\{C\}} = \Omega\{D_1 \ldots D_n\}$
$\Omega \in \{layer\ C, class\ C\ ext\ C'\}$
$1 \leq j_1 < j_k \leq n\ and\ \forall i : j_i < j_{i+1}$
$\forall i : D_{j_i} \in \textsf{METHOD}$

Recall that $X/_{\{q\}}$ is the domain restriction function [21, p. 50], which restricts a set of syntactic constructs X to the ones containing an identifier q.

Layers is the set of names of all layers in a program:

$$Layers : \textsf{PROGRAM} \longrightarrow \textsf{CLSNAME}^*$$

$$Layers(\varepsilon) = \phi$$
$$Layers(L :: P) = id(L) :: Layers(P)$$
$$Layers(Z :: P) = Layers(P)$$

where $P \in \textsf{PROGRAM}$, $L \in \textsf{LAYER}$ and $Z \in \textsf{CLASS}$.

Dealing With Layers

When calling a method m on a certain object, lookup involves more than just investigating the inheritance hierarchy of that object's dynamic type. Indeed, the method implementation found using the standard object-oriented lookup strategy may be superseded by an identically named method implementation in an active layer. Note that, for simplicity reasons, we do not take the possibility of overloaded methods into account.

The *layerMethod* function, given a PROGRAM, a method name, the name of its defining class and an ordered set of (globally unique) names of active layers, looks for the first applicable method implementation in one of these layers, and returns it together with the name of the layer where it was encountered, as shown in Fig. 15.

Operational Semantics

In the same style as *j* semantics, operational semantics for *cj* are expressed by means of reduction rules, which manipulate configurations consisting of a *cj* expression, a set of active layers, a heap and a stack frame. More precisely, semantics are defined by means of a function which calculates the value of an expression in the context of a certain program and a set of currently active layers. Additionally, the function returns a potentially modified heap, which models side effects:

$$\rightsquigarrow : \textsf{PROGRAM} \longrightarrow \textsf{EXP} \times Heap \times Stack \times \wp(\textsf{CLSNAME}) \longrightarrow Address \times Heap$$

The heap stores objects, which means it maps addresses to an array of field values, while the stack holds values for *this*, x, s and *thisLayer*.

Layer activation is done by means of the *withlayer* expression, for which the semantics are given below:

$$\frac{P \vdash e,h,s,\{L_1 \ldots L_n, L\} \rightsquigarrow \iota,h'}{P \vdash withlayer(L)\{e\},h,s,\{L_1 \ldots L_n\} \rightsquigarrow \iota,h'}$$

It simply appends the specified layer name L to the ordered set of currently active layers $\{L_1 \ldots L_n\}$, and evaluates the body of the *withlayer* expression.

Analogously, the *withoutlayer* expression allows evaluation of its body while temporarily deactivating a currently active layer:

$$\frac{P \vdash e,h,s,\mathscr{L} \backslash \{lyr \in \mathscr{L} | lyr = L\} \rightsquigarrow \iota,h'}{P \vdash withoutlayer(L)\{e\},h,s,\mathscr{L} \rightsquigarrow \iota,h'}$$

Activating and/or deactivating multiple layers can be achieved by nesting the appropriate expressions. Note that *withlayer* semantics allows the same layer to be activated multiple times, in which case *withoutlayer* semantics deactivates them all.

thisLayer

During execution of a layer method, it is possible for that method to access the singleton instance of its defining layer by means of the *thisLayer* variable. It can be used to access that layer's fields. In order to model this semantically, we assume a *cj* program always executes in the context of a prepared heap. More specifically, as a layer closely resembles a class syntactically, class instantiation semantics (cf. the *new* expression in Sec. 3.1 (9)) can be applied to a layer in order to obtain an object stored on the heap which maps the names of the layer fields to values. Moreover, the definition of the $Heap$ function is slightly extended in order to allow these singleton layer instances to be looked up by name:

$$Heap : (Address \rightarrow Object) \cup (\textsf{CLSNAME} \rightarrow Address)$$

$$layerMethod : \textsf{PROGRAM} \times Identifier \times \textsf{CLSNAME} \times \wp(\textsf{CLSNAME}) \rightarrow \textsf{METHOD} \times \textsf{CLSNAME}$$

$$\begin{array}{l} layerMethod(P,m,C,\varepsilon) = Undefined \\ layerMethod(P,m,C,\{L_1 \ldots L_k\}) = \\ \quad \left\{ \begin{array}{ll} (M,L_1) & if\ L_1 \in Layers(P)\ and\ \exists!i : getMethods(P,L_1)/_{\{m\}} = \\ & \qquad M_1 \ldots M_i \ldots M_n\ and\ M_i = T\ C.m(T'\ x)\{e\} \\ layerMethod(P,m,C,\{L_2 \ldots L_k\}) & otherwise \end{array} \right. \end{array} \tag{17}$$

Figure 15. *layerMethod* function, which looks for a method matching a given method name in an active layer.

The prepared heap for a *cj* program is therefore defined as follows:

$$\begin{array}{rl} h_P^{cj} = & \{L_1 \mapsto \iota_1, \ldots, L_n \mapsto \iota_n, \\ & \iota_1 \mapsto new^{\ell}(P,L_1), \ldots, \iota_n \mapsto new^{\ell}(P,L_n)\} \end{array} \tag{18}$$

where
$\forall i : L_i \in Layers(P)$
$\iota_1 \ldots \iota_n$ are unique

Note that this requires layer names to be globally unique.

Method Call Semantics

Determining which layers are currently active, and in which order their respective functionalities are to be applied, is obviously a very important task during method lookup. This is incorporated in the reduction rule shown in Fig. 16, expressing method call semantics.

As this rule is fundamental in order to understand *cj* semantics, we will explain it in more detail. The first line applies the semantics function recursively in order to obtain the address of the target object for the call. Note that $\mathscr{L}$ is the set of currently active layers, while e_0 can be any valid `EXP`, as far as its evaluation results in an address. In a similar way, the second line calculates the value of the actual parameter e_1. Note that the calculations so far might have caused side effects on the heap (e. g., if a subexpression of e_0 or e_1 involves field assignment), which is reflected by returning differently named heap variables.

The third line performs traditional object-oriented lookup, finding the method implementation to be executed in a context where no layers are active. The M_t^{ℓ} function is a slight variation of M^{ℓ} [21] which returns, apart from the relevant method implementation, also the name of the class where it was found. It is shown in Fig. 17.

The *type* and *super* functions are as in [21] and respectively determine the name of the dynamic type of an object and name of the super class of a class.

The fourth line of (19) uses the *layerMethod* function from Sec. 3.4 in order to determine whether there is a method implementation in a layer superseding the one found by M_t^{ℓ}. If there is no such method, M_2 will be undefined.

The next line invokes another auxiliary function called *actualMethod* to select which method should ultimately be executed. Basically, this is M_2, unless that was undefined, in which case M_1 is selected:

$$actualMethod : \textsf{METHOD} \times \textsf{METHOD} \rightarrow \textsf{METHOD}$$

$$actualMethod(M_1,M_2) = \left\{ \begin{array}{ll} M_1 & if\ M_2 = Undefined \\ M_2 & otherwise \end{array} \right.$$

In case M_1 is undefined as well, it is still selected, but execution will never happen since the result of *actualMethod* will not have the required form $T\ C.m(T'\ x)\{e\}$.

The last two lines, finally, are responsible for executing the body of the selected method. First, the appropriate stack variables are bound, including *msg*, which stores the current message in order to deal with *proceed* semantics later on. *thisLayer* and *s*, on the other hand, only have meaning in layer methods.

Ultimately, execution takes place by recursively applying the semantics function, passing the newly created stack frame. The result of this execution is returned as the result value of the complete reduction rule.

Proceed Semantics

If method lookup results in selecting a layer method for execution, the *proceed* expression may appear in its body, which should result in executing the next applicable layer method or (if there are none left) the method implementation selected by standard object-oriented lookup (M_1 in (19)). A different argument may be passed along.

Its semantics are specified by the reduction rule in Fig. 18. It looks very similar to the method call reduction rule, except that the set of active layers is not considered in its entirety, but merely those elements which appear after *thisLayer*, which denotes the method currently being executed. Furthermore, note that *this*, *s* and *msg* remain unchanged, while *x* and *thisLayer* are updated appropriately.

Mapping to the Machine Model

The main difference between *j* and *cj* lies in the latter's support for *layers,* which can be dynamically activated and de-

$$\frac{\begin{array}{c} P \vdash e_0, h, s, \mathscr{L} \rightsquigarrow \iota, h' \\ P \vdash e_1, h', s, \mathscr{L} \rightsquigarrow \iota', h'' \\ M_t^\ell(P, type(h'', \iota), m) = (M_1, C) \\ layerMethod(P, m, C, \mathscr{L}) = (M_2, L) \\ actualMethod(M_1, M_2) = T\ C.m(T'\ x)\{e\} \\ s' = \begin{cases} \{this \mapsto \iota, msg \mapsto m, x \mapsto \iota'\} & if\ M_2 = Undefined \\ \{\mathrm{s} \mapsto s(this), thisLayer \mapsto h''(L), this \mapsto \iota, msg \mapsto m, x \mapsto \iota'\} & otherwise \end{cases} \\ P \vdash e, h'', s', \mathscr{L} \rightsquigarrow \iota'', h''' \end{array}}{P \vdash e_0.m(e_1), h, s, \mathscr{L} \rightsquigarrow \iota'', h'''} \tag{19}$$

Figure 16. Reduction rule expressing method call semantics.

$$M_t^\ell : \textsf{PROGRAM} \times \textsf{CLSNAME} \times Identifier \rightarrow \textsf{METHOD} \times \textsf{CLSNAME}$$

$$M_t^\ell(P, C, m) = \begin{cases} Undefined & if\ C = \textsf{Object} \\ (M, C) & if\ getMethods(P, C)/_{\{m\}} = M \\ M_t^\ell(P, super(P, C), m) & if\ getMethods(P, C)/_{\{m\}} = \varepsilon \\ Undefined & otherwise \end{cases}$$

Figure 17. M_t^ℓ function.

$$\frac{\begin{array}{c} P \vdash e_0, h, s, \{L_1 \ldots L_n\} \rightsquigarrow \iota, h' \\ \exists i : type(h', s(thisLayer)) = L_i \\ M_t^\ell(P, type(h', s(this)), s(msg)) = (M_1, C) \\ layerMethod(P, s(msg), C, \{L_{i+1} \ldots L_n\}) = (M_2, L) \\ actualMethod(M_1, M_2) = T\ C.m(T'\ x)\{e\} \\ s' = \begin{cases} \{this \mapsto s(this), msg \mapsto s(msg), x \mapsto \iota\} & if\ M_2 = Undefined \\ s[thisLayer \mapsto h'(L)][x \mapsto \iota] & otherwise \end{cases} \\ P \vdash e, h', s', \{L_1 \ldots L_n\} \rightsquigarrow \iota', h'' \end{array}}{P \vdash proceed(e_0), h, s, \{L_1 \ldots L_n\} \rightsquigarrow \iota', h''} \tag{20}$$

Figure 18. Reduction rule expressing *proceed* semantics.

activated. Actually, code outside a *withlayer* block is executed strictly according to the *j* semantics. *withlayer*, however, activates a certain layer, causing the methods specified in that layer to shadow the corresponding class methods. *cj* semantics handle this by utilizing a number of helper functions analyzing the syntax and determining the applicable method, based on an ordered set of active layers.

Our machine model, on the other hand, once more takes advantage of the built-in delegation principle in order to establish the desired behavior. This requires a correct arrangement of delegation chains on the heap upon layer activation and deactivation.

At program start, no layers are active yet, so the heap should initially be prepared in the same way as (10) for *j* in Sec. 3.1. However, similarly to aspects in Sec. 3.3, layers support fields that may be accessed by layer methods. Hence, a singleton object for each layer is stored on the prepared heap as well, as shown in (18). In summary, the required prepared heap is intuitively the union of (10) and (18). We assume there to be no overlaps of their respective address spaces.

Upon activation of a certain layer via *withlayer*, a *layer proxy* is created for each layer method. The proxy contains that method's implementation, and is inserted in the delegation chain of the targeted class. An example of the resulting situation is displayed in Fig. 19. After executing the last instruction of the *withlayer* block, layer proxies are removed again. Hence, the delegation-based semantics of *withlayer* are as shown in Fig. 20, where $Lyr : Address \rightarrow \textsf{CLSNAME}$ is a function which is stored on the heap, and keeps track of which layer the proxies are associated with. This is necessary in order to be able to deactivate a certain layer, which should happen as a result of the *withoutlayer* expression, which may appear in a *withlayer* block. Its semantics are

$$
\begin{array}{c}
getMethods(P,L) = M_1 \ldots M_n \\
\forall i : M_i = T_i\ C_i.m_i(T_i'\ x)\{e_i\} \\
\forall k \in [1,n] : \iota_{lp,k} \notin dom(h) \\
h' = \quad h[\iota_{lp,1} \mapsto [\![m_1 : e_1]\!]][Del_{\iota_{lp,1}} \mapsto Del_{h(C_1)}][Del_{h(C_1)} \mapsto \iota_{lp,1}] \\
\ldots \\
[\iota_{lp,n} \mapsto [\![m_n : e_n]\!]][Del_{\iota_{lp,n}} \mapsto Del_{h(C_n)}][Del_{h(C_n)} \mapsto \iota_{lp,n}] \\
[Lyr(\iota_{lp,1}) \mapsto L] \ldots [Lyr(\iota_{lp,n}) \mapsto L] \\
P \vdash e, h', s \rightsquigarrow_\delta \iota, h'' \\
h''' = \quad h''[Del_{h(C_1)} \mapsto Del_{\iota_{lp,1}}] \ldots [Del_{h(C_n)} \mapsto Del_{\iota_{lp,n}}] \\
\hline
P \vdash withlayer(L)\{e\}, h, s \rightsquigarrow_\delta \iota, h'''
\end{array}
$$

Figure 20. Delegation-based semantics of *withlayer*.

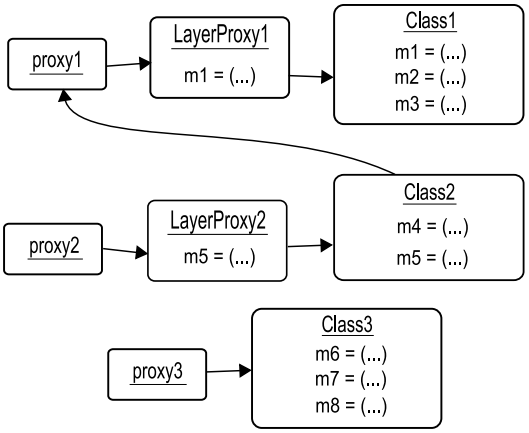

Figure 19. A layer which shadows method *m*1 in *Class*1 and *m*5 in *Class*2 has been activated. *Class*3 is not affected.

expressed by the following reduction rule:

$$
\begin{array}{c}
\{\iota_1, \ldots, \iota_n\} = \{\iota' \in dom(h) | Lyr(\iota') = L\} \\
\forall i : \exists! \iota_{C_i} : Del_{\iota_{C_i}} = \iota_i \\
h' = h[Del_{\iota_{C_1}} \mapsto Del_{\iota_1}] \ldots [Del_{\iota_{C_n}} \mapsto Del_{\iota_n}] \\
P \vdash e, h', s \rightsquigarrow_\delta \iota, h'' \\
h''' = h''[Del_{\iota_{C_1}} \mapsto \iota_1] \ldots [Del_{\iota_{C_n}} \mapsto \iota_n] \\
\hline
P \vdash withoutlayer(L)\{e\}, h, s \rightsquigarrow_\delta \iota, h'''
\end{array}
$$

Essentially, all proxies belonging to the specified layer are temporarily removed from the delegation chain of the class they apply to, and reinserted after execution of the *withoutlayer* block is finished.

By construction, the existing *Select* semantics (cf. Sec. 2 (6)) once more suffice to select the correct method implementation: If a method is shadowed by an active layer, the corresponding proxy will be in place, and it will intercept the message before it reaches its target. Nevertheless, similarly to Sec. 3.3, *Select* semantics do need a slight adaptation in order to correctly bind the stack variables *s*, which denotes the caller object, and *thisLayer*, which may be used in a layer method to access layer fields. The modified *Select* rule is shown in Fig. 21.

Note that, unlike *before* and *after* advice in Sec. 3.3, messages are not resent by default. In concordance with context-oriented programming principles, they completely shadow the corresponding class method, unless the layer method explicitly includes the *proceed* expression in its body. *cj* semantics for *proceed*, as defined by (20), use the *thisLayer* stack variable to determine which layer the current layer method pertains to. Next, an attempt is made to find a layer that also contains an applicable layer method, and that was activated later than the current layer. If successful, said method is executed. If not, the originally shadowed method is executed.

In the delegation-based model, however, this is, by construction, exactly the order in which the *Resend* rule (cf. Sec. 2 (7)) executes methods. Hence, *proceed* simply maps to *resend*:

$$
\frac{P \vdash resend(e), h, s \rightsquigarrow_\delta \iota, h'}{P \vdash proceed(e), h, s \rightsquigarrow_\delta \iota, h'}
$$

where the *Resend* rule has been slightly modified in order to correctly update *thisLayer*, as shown in Fig. 22.

Note that, although the *aj* language does not support *around* advice (cf. Sec. 3.3), it could be added straightforwardly using a similar *proceed* instruction, with essentially identical semantics.

4. Related Work

A substantial amount of other work deals with the formalization of aspect-oriented programming. Aspectual CAML [16] is an aspect-oriented extension to OCAML, with join points for typical functional features such as curried function calls and variant construction, as well as support for inter-type declarations. A compiler translates an Aspectual CAML program into an OCAML one, where function bodies are modified to call advice. This is similar to the original *ij* semantics by Skipper [21], where an *ij* program is translated into a *j* program.

The Aspect Sand Box [15] is an experimental workbench for different styles of aspect-oriented programming, centered on a small object-oriented base language, called BASE,

$$\frac{\begin{array}{c} e_0,h,s \rightsquigarrow_\delta \iota,h' \\ e_1,h',s \rightsquigarrow_\delta \iota',h'' \\ Look(h'',\iota,m) = (b,\iota_d) \\ s' = \begin{cases} \{\mathrm{s} \mapsto s(this), thisLayer \mapsto h''(L), this \mapsto \iota, msg \mapsto m, x \mapsto \iota', cur \mapsto \iota_d\} & if\ Lyr(\iota_d) = L \\ \{this \mapsto \iota, msg \mapsto m, x \mapsto \iota', cur \mapsto \iota_d\} & if\ Lyr(\iota_d) = Undefined \end{cases} \\ P \vdash b,h'',s' \rightsquigarrow_\delta \iota'',h''' \end{array}}{P \vdash e_0.m(e_1),h,s \rightsquigarrow_\delta \iota'',h'''}$$

Figure 21. Modified *Select* rule, ensuring correct binding of stack variables.

$$\frac{\begin{array}{c} e,h,s \rightsquigarrow_\delta \iota,h' \\ Look(h',Del_{s(cur)},s(msg)) = (b,\iota_d) \\ s' = \begin{cases} s[thisLayer \mapsto h'(L)][x \mapsto \iota][cur \mapsto \iota_d] & if\ Lyr(\iota_d) = L \\ s[x \mapsto \iota][cur \mapsto \iota_d] & otherwise \end{cases} \\ P \vdash b,h',s' \rightsquigarrow_\delta \iota',h'' \end{array}}{P \vdash resend(e),h,s \rightsquigarrow_\delta \iota',h''}$$

Figure 22. Modified *Resend* rule.

which is extended with aspect-oriented features. Wand *et al.* [24] consider a procedural subset of BASE in order to investigate the formal semantics of a number of aspect-oriented features, including dynamic join points, pointcut designators and *before*, *after* and *around* advice. A monadic semantics is used, which explicitly models a weaving approach: At each procedure call, the weaver is invoked, taking a list of advice and a join point. The weaver determines which advice are applicable, and results in a new procedure, wrapping the original procedure in all of the applicable advice.

The Common Aspect Semantics Base [8] takes a similar approach, but applies a two-stage function at every join point, first checking whether the current instruction should be advised, and then verifying whether the present dynamic state calls for advice application.

Clifton and Leavens [4] introduce MiniMAO$_0$, a small object-oriented language for which an operational semantics is provided. The aspect-oriented MiniMAO$_1$ extension mimicks AspectJ's semantics of around advice on call and execution join points. Advice invocation occurs in several steps, essentially modeling a weaver: join points are created explicitly for each call, a list of applicable advice is searched for and finally evaluated. Interestingly, the authors state that they regard advice binding to be a primitive operation, similar to the object-oriented virtual dispatch mechanism. In our delegation-based semantics, however, advice invocation actually becomes part of virtual dispatch, every message send being a join point implicitly.

In [1], Aldrich introduces TinyAspect, a language with pointcuts and around advice. In the formal semantics, function calls are replaced by calls to applicable advice, where occurences of proceed are replaced by calls to the original function.

These explicit weaving approaches, whether they statically generate an aspect-free program, or dynamically invoke advice lookup strategies, have the disadvantage that aspect-oriented features are implemented as add-ons to a base which cannot deal with them directly. In fact, a weaving approach is encountered as well in the semantics of *aj* as defined originally by Skipper [21], as can be derived from Fig. 10 in Sec. 3.3, which also performs explicit advice lookup. Our work, however, shows that such helper constructs can be avoided by mapping mechanisms such as advice lookup onto a machine model, which solely relies on delegation and message sending, and reuses the existing dispatch strategy. Call join points, however, are currently not supported by our model.

Walker *et al.* [23] also use an intermediate model with inherent support for aspects. More specifically, their aspect-oriented core calculus is an extension of lambda calculus, which adds labeled join points and advice at those join points. Advice invocation is handled once more explicitly, as a list of advice is checked at each named join point and a composed function of all applicable advice is created and called. A language MinAML, which distinguishes between *before*, *after* and *around* advice, is then introduced, in order to show that these features can be translated to the core calculus. To this end, labeled join points are explicitly introduced at function entry and exit, where *before* advice applies to the former, and *after* advice to the latter. *around* advice is handled by means of a goto mechanism which allows jumping directly to a labeled join point. Ultimately, all this means that the explicit advice invocation mechanism is still present in the MinAML semantics. In [7], the AspectML language is translated to the core calculus in a similar way. In contrast to our work, the core calculus of Walker *et al.* does include

a type system, which is considered future work here. Additionally, complex pointcut specifications, including *cflow*, are supported, while we did not yet cover a language exhibiting those features, although our machine model itself has the ability to handle *cflow* [12].

Lämmel [14] observes that method call interception (MCI), which can be regarded as a more explicit form of delegation, is a powerful construct which can capture several aspect-oriented features. However, he proposes to introduce this as a language construct, at the same level as (virtual) method invocation. This is different from our work, where delegation is part of the machine model, but not necessarily of high-level aspect-oriented languages which are translated to it.

Composition filters [3] is another approach which exploits the language level, rather than the machine level. Here, outgoing as well as incoming messages pass through a number of filters, which may for example reroute dispatch. Delegation is mentioned as a possible application.

In summary, our approach is unique in the sense that semantics of aspect-oriented features are expressed in terms of a machine model, which seamlessly incorporates aspect-orientation as opposed to explicitly modeling advice invocation. While this is clearly beneficial to understandability, another advantage of our approach is in the implementation of an aspect-oriented virtual machine, which may be simplified as well as explicit helper mechanisms such as advice lookup can be avoided.

Implementation-wise, a number of projects are concerned with virtual machine support for aspect-orientation [19, 20, 11, 10, 22]. Among these, PROSE 2 [20] is the only one to adopt, like our machine model, a view on the running application that explicitly regards *all* points in the execution as potential join points *by default*. This is realized by instrumenting the implementation to call an AOP infrastructure at each potential join point, and having the infrastructure explicitly check for the applicability of advice. Our model, conversely, is based on *implicitly* executing advice by means of message interception and dynamically updating delegation chains.

5. Summary and Future Work

We have presented semantic mappings from four high-level programming languages, *j*, *ij*, *aj* and *cj* to our previously introduced machine model. Additionally, we explained informally that our model correctly supports the various mechanisms which are included in these languages in order to address crosscutting concerns. Hence, the model is shown to be sufficiently expressive to meet the requirements of different languages with different approaches to modularization. Moreover, as the semantics used in this paper are operational, the mappings suggest how an implementation of these languages based on our model can be derived. A prototypical implementation of all four languages on top of an implementation of the model has indeed been realized as well. The paper has introduced a formal semantics for *cj*, a language supporting context-oriented programming, and essentially a subset of ContextJ.

There are several areas of future work. The semantic mappings presented here need to be extended with features such as parameter overloading and type soundness. To that end, the formal semantics of our model needs to be extended with type information. Additionally, a formal proof will be provided for the claim that the semantic mappings are behavior preserving. A proof-of-concept implementation of our model is already in place, and the languages for which semantics have been presented above have been implemented using this implementation. The focus in devising these language implementations has been on faithfully adopting their semantics instead of providing high performance. Whereas this work shows that the language mappings can be done correctly, we intend to demonstrate that this can be done in an efficient manner as well. Dedicated caching mechanisms to deal with message lookup, and garbage collection strategies to deal with the large amount of small (proxy) objects, are a worthwhile path to investigate. The model itself is being investigated further, especially regarding the support of additional constructs such as call join points.

References

[1] Jonathan Aldrich. Open Modules: A Proposal for Modular Reasoning in Aspect-oriented Programming. In *Foundations of Aspect Languages*, 2004.

[2] Christopher Anderson and Sophia Drossopoulou. δ – An Imperative Object-based Calculus with Delegation. In *Proc. USE'02*, Malaga, 2002.

[3] Lodewijk Bergmans and Mehmet Aksit. Composing Crosscutting Concerns Using Composition Filters. *Commun. ACM*, 44(10):51–57, 2001.

[4] Curtis Clifton and Gary T. Leavens. MiniMAO1 – An Imperative Core Language for Studying Aspect-oriented Reasonings. *Sci. Comput. Program.*, 63(3):321–374, 2006.

[5] Pascal Costanza and Robert Hirschfeld. Language Constructs for Context-oriented Programming: An Overview of ContextL. In *Dynamic Languages Symposium (DLS) '05, co-organised with OOPSLA'05*. ACM Press, 2005.

[6] Pascal Costanza, Robert Hirschfeld, and Wolfgang De Meuter. Efficient Layer Activation for Switching Context-dependent Behavior. In *JMLC*, pages 84–103, 2006.

[7] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Trans. Program. Lang. Syst.*, 30(3):1–60, 2008.

[8] Simplice Djoko Djoko, Remi Douence, Pascal Fradet, and Didier Le Botlan. CASB: Common Aspect Semantics Base. Technical Report AOSD-Europe Deliverable D41, AOSD-Europe-INRIA-7, INRIA, France, 10 February 2006.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John

Vlissides. *Design Patterns — Elements of Reusable Object-oriented Software*. Addison-Wesley, 1994.

[10] Michael Haupt. *Virtual Machine Support for Aspect-oriented Programming Languages*. PhD thesis, Software Technology Group, Darmstadt University of Technology, 2006.

[11] Michael Haupt, Mira Mezini, Christoph Bockisch, Tom Dinkelaker, Michael Eichberg, and Michael Krebs. An Execution Layer for Aspect-oriented Programming Languages. In *Proc. VEE 2005*. ACM Press, June 2005.

[12] Michael Haupt and Hans Schippers. A Machine Model for Aspect-oriented Programming. In *ECOOP 2007 - Object-oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 501–524. Springer, 2007.

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

[14] Ralf Lämmel. A Semantical Approach to Method-call Interception. In *Proc. AOSD'02*, pages 41–55. ACM Press, 2002.

[15] Hidehiko Masuhara and Gregor Kiczales. Modeling Crosscutting in Aspect-oriented Mechanisms. In Luca Cardelli, editor, *ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 2–28. Springer, 2003.

[16] Hidehiko Masuhara, Hideaki Tatsuzawa, and Akinori Yonezawa. Aspectual Caml – An Aspect-oriented Functional Language. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 320–330, New York, NY, USA, 2005. ACM.

[17] Todd D. Millstein and Craig Chambers. Modular Statically Typed Multimethods. In *ECOOP '99: Proceedings of the 13th European Conference on Object-oriented Programming*, pages 279–303, London, UK, 1999. Springer-Verlag.

[18] Harold Ossher. A Direction for Research on Virtual Machine Support for Concern Composition. In *Proc. Workshop VMIL '07*. ACM Press, 2007.

[19] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic Weaving for Aspect-oriented Programming. In Gregor Kiczales, editor, *Proc. AOSD 2002*. ACM Press, 2002.

[20] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Just-in-Time Aspects. In *Proc. AOSD 2003*. ACM Press, 2003.

[21] Mark C. Skipper. *Formal Models for Aspect-oriented Software Development*. PhD thesis, Imperial College, London, 2004.

[22] Eric Tanter and Jacques Noyé. A Versatile Kernel for Multi-Language AOP. In *Proc. GPCE'05*. Springer, 2005.

[23] David Walker, Steve Zdancewic, and Jay Ligatti. A Theory of Aspects. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 127–139, New York, NY, USA, 2003. ACM.

[24] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-oriented Programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.

A. Errata

Sec. 2.2 (1): Syntax construct ';' was not introduced, should be substituted:

$$\frac{\begin{array}{c} a,\sigma \rightsquigarrow_\delta \iota,\sigma' \\ \iota' \notin dom(\sigma') \\ \sigma'' = \sigma'[\iota' \mapsto \sigma'(\iota)][Del_{\iota'} \mapsto Del_\iota] \\ \iota'' \notin dom(\sigma'') \\ \sigma''' = \sigma''[\iota'' \mapsto [\![]\!]][Del_{\iota''} \mapsto \iota'] \end{array}}{clone(a),\sigma \rightsquigarrow_\delta \iota'',\sigma'''}$$

Fig. 5: '$\mapsto$' is more appropriate than '$=$':

$$\begin{array}{rl} h_P^j = & \{C_1 \mapsto \iota_{proxy,1},\ldots,C_n \mapsto \iota_{proxy,n}, ClPt(C_1) \mapsto \iota_1,\ldots ClPt(C_n) \mapsto \iota_n, \\ & \iota_1 \mapsto [\![f_{1,1} : \text{null}\ldots f_{1,k_1} : \text{null}]\!],\ldots,\iota_n \mapsto [\![f_{n,1} : \text{null}\ldots f_{n,k_n} : \text{null}]\!], \\ & \iota_{proxy,1} \mapsto [\![]\!],\ldots,\iota_{proxy,n} \mapsto [\![]\!], \\ & \iota_{meth,1} \mapsto [\![m_{1,1} : b_{1,1},\ldots,m_{1,p_1} : b_{1,p_1}]\!],\ldots,\iota_{meth,n} \mapsto [\![m_{n,1} : b_{n,1},\ldots,m_{n,p_n} : b_{n,p_n}]\!], \\ & Del_{\iota_1} \mapsto \iota_{proxy,1},\ldots,Del_{\iota_n} \mapsto \iota_{proxy,n}, \\ & Del_{\iota_{proxy,1}} \mapsto \iota_{meth,1},\ldots,Del_{\iota_{proxy,n}} \mapsto \iota_{meth,n}, \\ & Del_{\iota_{meth,1}} \mapsto h_P^j(super^\ell(P,C_1)),\ldots,Del_{\iota_{meth,n}} \mapsto h_P^j(super^\ell(P,C_n))\} \end{array}$$

where

$\forall i : C_i \in Classes(P), AllFields(P,C_i) = f_{i,1}\ldots f_{i,k_i}, \forall j : T\ m_{i,j}(T'\ x)\{b_{i,j}\} \in getMethods(C_i)$

$ClPt : \textsf{CLSNAME} \rightarrow Address$

$\iota_1\ldots\iota_n, \iota_{proxy,1}\ldots\iota_{proxy,n}, \iota_{meth,1}\ldots\iota_{meth,n}$ are unique

p. 10, col. 2, l. 3: As $deploy(proxy,target)$ inserts a proxy in the delegation chain immediately after the target object
⇒ As proxies are inserted in the delegation chain immediately after the target object

p. 12, *withlayer*: In order to align with delegation-based semantics, prepend instead of append layers:
It simply appends ⇒ It simply prepends

$$\frac{P \vdash e,h,s,\{L,L_1\ldots L_n\} \rightsquigarrow \iota,h'}{P \vdash withlayer(L)\{e\},h,s,\{L_1\ldots L_n\} \rightsquigarrow \iota,h'}$$

Fig. 20: It is safer to rewire the *Del* function in reverse order at the end:

$$\frac{\begin{array}{c} getMethods(P,L) = M_1\ldots M_n \\ \forall i : M_i = T_i\ C_i.m_i(T_i'\ x)\{e_i\} \\ \forall k \in [1,n] : \iota_{lp,k} \notin dom(h) \\ \begin{array}{rl} h' = & h[\iota_{lp,1} \mapsto [\![m_1 : e_1]\!]][Del_{\iota_{lp,1}} \mapsto Del_{h(C_1)}][Del_{h(C_1)} \mapsto \iota_{lp,1}] \\ & \ldots \\ & [\iota_{lp,n} \mapsto [\![m_n : e_n]\!]][Del_{\iota_{lp,n}} \mapsto Del_{h(C_n)}][Del_{h(C_n)} \mapsto \iota_{lp,n}] \\ & [Lyr(\iota_{lp,1}) \mapsto L]\ldots[Lyr(\iota_{lp,n}) \mapsto L] \end{array} \\ P \vdash e,h',s \rightsquigarrow_\delta \iota,h'' \\ h''' = \ h''[Del_{h(C_n)} \mapsto Del_{\iota_{lp,n}}]\ldots[Del_{h(C_1)} \mapsto Del_{\iota_{lp,1}}] \end{array}}{P \vdash withlayer(L)\{e\},h,s \rightsquigarrow_\delta \iota,h'''}$$

p. 15, *withoutlayer*: It is safer to rewire the *Del* function in reverse order at the end:

$$\frac{\begin{array}{c} \{\iota_1,\ldots,\iota_n\} = \{\iota' \in dom(h) | Lyr(\iota') = L\} \\ \forall i : \exists!\iota_{C_i} : Del_{\iota_{C_i}} = \iota_i \\ h' = h[Del_{\iota_{C_1}} \mapsto Del_{\iota_1}]\ldots[Del_{\iota_{C_n}} \mapsto Del_{\iota_n}] \\ P \vdash e,h',s \rightsquigarrow_\delta \iota,h'' \\ h''' = h''[Del_{\iota_{C_n}} \mapsto \iota_n]\ldots[Del_{\iota_{C_1}} \mapsto \iota_1] \end{array}}{P \vdash withoutlayer(L)\{e\},h,s \rightsquigarrow_\delta \iota,h'''}$$

p. 15, col. 2, l. 10: that was activated later than the current layer ⇒ that was activated prior to the current layer