Compositionality of Aspect Weaving

Florian Kammüller and Henry Sudhof

Technische Universität Berlin
Insititut für Softwaretechnik und Theoretische Informatik

Abstract. One approach towards adaptivity is aspect-orientation. Aspects enable the systematic addition of code into existing programs. In order to provide safe and at the same time flexible aspects for such adaptive systems we address the verification of the aspect-oriented language paradigm. This paper first gives an overview of our aspect calculus and summarizes previous results. Then we present a new compositionality lemma prerequisite for so-called run-time weaving. The entire theory and proofs are carried out in the theorem prover Isabelle/HOL.

1 Introduction

An important research subject concerning networks of highly autonomous components in distributed systems is *adaptability*. Systematic code adaptation is a necessity for the reliability and the deployment on a large scale of autonomous component systems. Adaptability requires that new artifacts may be dynamically added as a prerequisite for deployment or quality of service requirements of an instance, but also during the entire life cycle of a software component. Moreover, the realization of partially autonomous components, that can, for example, defend themselves automatically against attacks, demands as well that the notion of adaptability is well understood, easily implemented and modelled. By providing a set of aspects a toolset can be constructed from which autonomous components can select aspects to adapt to current needs.

Automated formal analysis with proof assistants provides a strong support for the analysis of safety properties of programming languages [6]. Our approach to support the verification of adaptive systems consists of providing a fully formalized basis for aspect-oriented programming in Isabelle/HOL [4]. We construct a core calculus of objects and aspects with types as an instance of the generic theorem prover Isabelle/HOL. The resulting framework serves to experiment with language features – for example, weaving functionality and pointcut selectors – and properties – for example, type safety and compositionality. At the same time, our results have mathematical precision and are mechanically verified. Moreover, we try to keep the formal model of the aspect calculus as constructive as possible. Thereby, we can extract executable prototypes for evaluators and type checkers from the Isabelle/HOL framework. In this paper we first give in Section 2 a short introduction to necessary prerequisites for our work. In Section 3 we present our calculus of aspects summarizing previous results on confluence and type safety. Section 4 then presents in detail the compositionality theorem for ς_{Asc}, followed by Section 5 closing with a discussion.

2 Preliminaries

Aspects enable the systematic and efficient adaptation of existing programs by adding (weaving) code segments (advice) at user-defined points (pointcuts). For example, a given implementation for a secure group communication in a network could be adapted to support only authenticated channels by weaving in an advice that implements user authentication prior to any remote method call.

Our DFG-funded project Ascot [4] mechanizing aspect-orientation has produced some important first results [5, 7–9] forming a sound basis for security-critical applications of aspect-orientation. More specifically, we have constructed a full formalization of the ς-calculus in Isabelle/HOL, proved confluence, and extended the base calculus to ς_{Asc}, a calculus for aspects and weaving. More prominently, we have defined a type system for aspects on ς_{Asc} and proved type safety [8]. The basic idea of our calculus of aspects is similar to the theory of aspects [10] but we start from the Theory of Objects, unlike the former that is based on the λ-calculus. To model aspects we introduce labels in the base program. These labels represent so-called join-points, i.e. points at which advice might be woven in. Given these labels, we can quite naturally define weaving: advice is given as a function applicable to a labelled term, replacing the original term. So, given an aspect as a pair of pointcuts L and an advice that shall be applied at all join-points specified by L, weaving can be constructed by function application, as illustrated in the following example (weaving is represented as ⇓).

$$\langle L.\lambda\ x.\ e\rangle \Downarrow (v_1 + l_1\langle v_2\rangle) \xrightarrow{l_1 \in L} v_1 + e[v_2/x]$$

We next introduce the prerequisites for the ς_{Asc} calculus.

2.1 The ς-calculus

In a *Theory of Objects*[1] Abadi and Cardelli developed the ς-family of calculi to formally study object-orientation. These calculi are widely accepted as conceptual equivalents of the λ-calculus for objects, since the objects can be directly used as a basic construct without having to be simulated through λ-expressions. In the ς-calculi, an object is defined as a set of labelled methods. Each method is a ς-term in its own right and has a parameter *self*, in which the enclosing object is contained. There are three flavors of primitives from which to build such terms: *object definitions*, *method invocation* and *field update*, which are presented in Figure 1. Methods not using the self parameter are considered to be *fields.*

Let $o \equiv [li = \varsigma(x_i)bi^{i\in 1..n}]$ (l_i distinct)
o is an object with method names l_i and methods $\varsigma(x_i)b_i$
$o.l_j \rightarrow b_j\{x_j \leftarrow o\}(j \in 1..n)$ selection / invocation
$o.l_j \Leftarrow \varsigma(y)b \rightarrow [l_j = \varsigma(y)b, l_i = \varsigma(xi)b_i^{i\in(1..n)-j}](j \in 1..n)$ update / override

Fig. 1. The primitive semantics of the ς-calculus as introduced in [1]

2.2 Isabelle/HOL

Isabelle [11] is an interactive ML-based theorem prover. It was initially developed by Lawrence Paulson at the University of Cambridge and is today maintained there and at the TU Munich. Unlike many other interactive provers, Isabelle was written to serve as a framework for various logics, so-called object-logics. Today, mostly the object-logic for Higher-Order-Logic (HOL) and – on a smaller scale – the one for Zermelo-Fraenkel set theory are in widespread use. Isabelle has a meta-logic serving as a deductive framework for the embedded object-logics. This meta-logic is itself a fragment of HOL solely consisting of the universal quantifier and the implication. Isabelle features a powerful simplifier, and automated proof strategies. For this paper, Isabelle/HOL was used, e.g. Isabelle in its instantiation to HOL. In Isabelle/HOL automatic code generation is possible for constructive parts of a formalization, like datatypes and inductive definitions (see below), but also for constructive proofs.

A very generic parser enables application-specific definition of concrete syntax (so called mixfix syntax) making Isabelle formulae and proofs almost identical to pen-and-paper formalizations. In general, Any Isabelle/HOL specific syntax that we will be using throughout the paper is going to be explained when we use it.

2.3 De Bruijn Indices

One known hard problem [12] in the formalization of language semantics is the representation of *binders*, e.g. the operator λ in the λ-calculus that binds a variable x over a term t in which x may occur. More precisely, the actual problem lies in the complexity created by isomorphic terms that differ only in the choice of variable names: α-conversion.

De Bruijn indices overcome the problem of concrete variable names, and thus α-conversion, by simply eliminating them. A variable is replaced by a natural number that represents the distance — in terms of nesting depth — of this variable to its binder. Thereby terms contain only numbers, no variable; α-conversion becomes obsolete. This is a considerable advantage as α-conversion is a difficult problem both from a practical point of view and for mechanical proofs. An example for illustrating the use of de Bruijn indices is given by the following simple λ-term.

$$\lambda x.\lambda y.(\lambda z.\, x\, z)y = Abs(Abs(Abs(Var\, 2)\$(Var\, 0))(Var\, 0))$$

Note that, different variables may be represented by the same number, e.g., z and x both are $Var\, 0$. De Bruijn indices relieves one from having to deal with α-conversion: for example both $\lambda x.x$ and its α-equivalent $\lambda y.y$ are represented by $Abs(Var 0)$. The disadvantage of de Bruijn indices is that substitution, normally used for the definition of application, is difficult to construct. A term has to be "lifted", that is, his "variables" have to be increased by one, when it moves into the scope of an abstraction in the process of substitution. We will see these operations when we introduce our ς_{Asc}-calculus in the next section.

3 A Theory of Aspects in Isabelle/HOL

3.1 Terms of the ς_{Asc} Calculus

```
datatype dB =  Var nat
            |  Obj (label ⇀ dB) type
            |  Call dB label
            |  Upd dB label dB
            |  Asp Label dB    ("_ ⟨ _ ⟩")
```

The constructor `Var` builds-up a new term `dB` from a `nat` representing the de Bruijn index of the variable. An object is recursively defined by a finite map from `label`, the predefined types of "field names", to arbitrary terms of type `dB`. The second argument of type `type` to the `dB`-constructor `Obj` is the Object's type. We insert the type with an object in order to render the typing relation unique. The cases `Call` and `Update` similarly represent, field selection and update of an object's field. The field constructor `Asp` enables the insertion of aspect labels into object terms. These labels will not be assigned any semantics until the point where we define weaving in Section 3.5. The annotation behind the constructor in quotation marks defines the mixfix syntax: we can use the notation `l⟨t⟩` as abbreviation for `Asp l t`.

3.2 Lifting and Substitution

As de Bruijn indices discard the use of formal parameters, substitution has to be performed by adapting the numbers representing variables when a term is moved between different layers of the nested scopes of abstraction. This movement occurs precisely when a variable has to be substituted by a term containing a free variable inside the scope of an abstraction. Therefore the notion of substitution is chained with the notion of lifting. We declare the following two constants in Isabelle/HOL. We define two operators `lift` and `subst` using mixfix syntax again to write `t[s/n]` to express that in a term `t` the variable represented by `n` shall be replaced by `s`. Before defining the semantics of substitution we need to define the lifting of a term. A lifting carries a parameter `n` representing the *cut* between free and bound variable numbers in the term that shall be lifted. The operation `lift` is defined by the following set of primitive recursive equations describing the effect of lifting over the various cases of object terms.

```
  liftVar:    lift (Var i) k = (if i < k then Var i else Var (i + 1))
  liftObj:    lift (Obj f) k = (Obj (map (λ x. lift x (k + 1)) f))
  liftCall:   lift (Call a l) k = Call (lift a k) l
  liftUpd:    lift (Upd a l b) k = Upd (lift a k) l (lift b (k + 1))
```

A variable is only lifted when it is free, i.e. when its representing number is greater or equal to the "cut" parameter. The "cut" parameter is increased in the recursive call when an abstraction scope is entered. This is the case when the lift function enters inside a method in an object, and when a field is updated by a method. Note that we increase only on the right side of an update because the left side will always be an object seen as a reference whereas the right side is a method.

Based on the definition of lift, substitution can be defined as follows.

```
subst_Var:   Var i [s/k] =
              if k < i then Var (i - 1) else if i = k then s else Var i
subst_Obj:   Obj f [s/k] = Obj (map (λ x. x[(lift s 0)/(k+1)]) f)
subst_Call:  Call a l [s/k] = Call (a [s/k]) l
subst_Upd:   Upd a l b [s/k] = Upd (a [s/k]) l (b [lift s 0 / k+1])
```

The idea is that a term `s` is lifted if it is substituted inside an abstraction scope, i.e., inside an object and at the right side of an update. The lifting is always initiated with “cut” parameter `0` as initially the outermost free variable when entering a scope. The decrementing in the equation for `Var` in cases of free variables greater than the “cut” parameter is necessary to cancel out the previous effects of lifting.

3.3 Evaluation of Terms

Given the Isabelle/HOL definition of substitution for ς-terms as `t[s/n]` meaning *substitute* ***n*** *by* ***s*** *in* ***t*** using mixfix syntax, we define a small step operational semantics by a relation $\rightarrow_\beta$ using an inductive definition.

```
inductive →β
intros
  beta: l ∈ dom f  f ⟹ Call (Obj f) l →β the(f l)[(Obj f)/0]
  upd : l ∈ dom f  ⟹ Upd (Obj f T) l a →β  Obj (f (l ↦ a) T)
  sel : s →β t ⟹ Call s l →β Call t l
  updL: s →β t ⟹ Upd s l u →β Upd t l u
  updR: s →β t ⟹ Upd u l s →β Upd u l t
  obj : ⟦ s →β t; l ∈ dom f ⟧
        ⟹ Obj (f (l ↦ s) T)  →β Obj (f (l ↦ t) T)
  asp : s →β t ⟹ l ⟨ s ⟩ →β l ⟨ t ⟩
```

The rules represent quite closely the original semantics of ς. The substitution `[(Obj f T)/0]` in the rule `beta` replaces the self parameter for the outermost variable in the object's *lth* field `f l`. The operator `the` selects an α-element in an option datatype when it is defined, i.e. unequal to `None`. There is no case for labels because the semantics is attached to labels later by weaving.

3.4 Aspects

An aspect can be simply defined as a selection of pointcuts and an advice. Since our model is in Higher Order Logic, where sets are isomorphic to predicates, we can assume that our selection of pointcuts is a set of labels. The advice is a ς-term not enclosed in an object, because an advice is applied to the subexpression of a ς-program that is marked by a label returning another ς-term as a result. Hence, in Isabelle/HOL aspects can be simply defined as follows.

```
datatype aspect = Aspect (Label list) dB    ("⟨ _, _ ⟩")
```

The first element is the pointcut set `L` and the second element the advice to be applied to all points matching the pointcut description, i.e. being member of `L`. The mixfix syntax at the righthandside enables to annotate an aspect as ⟨`L, a`⟩.

3.5 Weaving

Given a base program in the ς-calculus readily labelled with aspect labels and given some aspects, the weaving function now only has to step through the term while applying the aspect. We consider this approach to resemble static weaving, but given the functional nature of our calculus, we consider the result to be valid for dynamic approaches as well. Therefore we define a function "weave" represented as ⇓ that takes a ς-program and an aspect and returns a ς-program. The second operator `weave_option` is an auxiliary function that is needed to "map" the weaving function over the finite maps representing objects.

```
  weave :: [ dB, aspect ] ⇒ dB    ("⇓")
  weave_option :: [ dB option, aspect ] ⇒ dB option ("⇓opt")
```

We define the weaving function for the simple case of applying one aspect to a program. The general case is later derived by repeated application. The definition of the simple case is given below in a mutual recursive definition defining the semantics of `weave` and `weave_option` by simple equations. In case of weaving an aspect onto a variable `Var n` the advice has no effect. The case `l⟨t⟩` is the interesting one because now the ς-term for aspects, `Asp`, is finally equipped with semantics. In case that the label is in the pointcut specified by the first component of the aspect then the aspect matches. Consequently the advice part of the aspect `a` is applied to the current term `t`. Otherwise the aspect has no effect. The label is not eliminated during the weaving process to enable repeated weaving.

```
primrec
   (Var n) ⇓ ⟨L, a⟩ = Var n
   l ⟨ t ⟩ ⇓ ⟨L.a⟩ = if  l ∈ set(L) then l ⟨ a[(t ⇓ ⟨L, a⟩)/0] ⟩
                                     else l ⟨ t ⇓ ⟨L, a⟩ ⟩
```

The Isabelle/HOL projection `set` transforms a list (here of labels) into the set of all elements contained in the list.

The next two equalities for `Call` and `Upd` simply define that the weave process is to be passed through to the corresponding sub-terms.

```
   (Call s l) ⇓ A = Call (s ⇓ A) l
   (Upd s l t) ⇓ A = Upd (s ⇓ A) l (t ⇓ A)
```

The primitive recursive equations defining the semantics for `Obj` is now the point where the recursion changes to the auxiliary operator `weave_option`. The auxiliary operator enables the pointwise definition of advice on the fields of the object by lifting the weaving function over the λ to argument position. In the defining equations for `weave_option` ($\Downarrow_{\text{opt}}$) we see the benefit gained by using the option type: we can explicitly use pattern matching to distinguish the case for unused field labels (`None`) and actual object fields matching out the field value with `Some`.

```
   (Obj f T) ⇓ A = Obj (λ l. ((f l) ⇓opt A)) T
   None ⇓opt A = None
   (Some t) ⇓opt A = Some (t ⇓ A)
```

4 Compositionality and Run-Time Weaving

An important question for aspects and their practical usability is the compositionality of weaving. In aspect parlance compositionality corresponds to the possibility of *run-time weaving*. Figure 2 illustrates this question for AspectJ graphically: when does this diagram commute? (index *sc* stands for source, *bc* for bytecode, *p* for program, and *ptc* and *adv* for pointcut and advice.) In more foundational terms this questions represents that aspect weaving is respected by compilation or evaluation of a program. Alternatively, we can say that aspect weaving is compositional.

$$\begin{tikzcd}
(p_{sc}, ptc_{sc}, adv_{sc}) \arrow[rr, "\texttt{weave\_sc}"] \arrow[d, "\langle \texttt{comp}, ptc_{comp}, \texttt{comp} \rangle"] & & p'_{sc} \arrow[d, "\texttt{comp}"] \\
(p_{bc}, ptc_{bc}, adv_{bc}) \arrow[rr, "\texttt{weave\_bc}"] & & p'_{bc}
\end{tikzcd}$$

Fig. 2. do compile-time and run-time weaving commute?

We have made a major step forward in tackling the compositionality question by proving the following central lemma.

Lemma 1 (Distribution of weave over subst). *Let* `a` *be a well-formed aspect, i.e. containing just one free variable, and* `s` *be a closed subterm (no free variables) of program* `t`*. Then weaving distributes over substitution.*

$$\texttt{justoneFV(adv a)} \wedge \texttt{noFV s} \implies \texttt{t[s/n]} \Downarrow \texttt{a = (t} \Downarrow \texttt{a)[(s} \Downarrow \texttt{a)/n]}$$

Considering the main rule `beta` of the operational semantics in Section 3.3, i.e. `Call (Obj f) l` $\rightarrow_\beta$`the(f l)[(Obj f)/0]` we see the significance for compositionality. This substitution represents function application in the language; weaving is based on function application as well. Hence, the lemma defines that weaving distributes over function application which is the essence of compositionality.

5 Conclusions

We have motivated the use of aspects for self-adapting systems and argued that a sound basis is prerequisite for safely constructing such systems. After an overview of our mechanized calculus ς_{Asc} for aspects we have addressed compositionality.

Compositionality is a very central property for any kind of software system because only compositionality enables to apply divide-and-conquer style of construction which is at the basis of most algorithmic solutions. Compositionality is often very hard to get by. Many properties of interest, like security for example, are not compositional – hence, the importance of our small result. We showed a central lemma that clears up the distributivity between function application, i.e. substitution, and weaving, the main drivers of the operational semantics of the ς_{Asc} calculus. In general, we are aiming at using a mechanized framework like Isabelle/HOL to build a sound core calculus with as much good properties as possible. We want to increase the understanding of the basic principles of aspect-orientation through adding stepwise specific constructs and thereby defining precise borderlines.

We have used de Bruijn indices in the formalization of our aspect calculus. As a final thought of this paper we would like to contemplate their significance. De Bruijn indices are often criticized because they differ from the intuitive notion of names. Recent approaches on nominal techniques [13] offer alternative techniques that are unfortunately not yet sufficiently developed for our application. However, there are other very recent techniques, like *locally-nameless*, that are based on de Bruijn indices and seemingly simpler to apply. We believe that certain results are easier proved with de Bruijn indices. For example, the compositionality result is quite easily derived with de Bruijn indices.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects.* Springer, New York, 1996.
2. H. P. Barendregt. *The Lambda Calculus, its Syntax and Semantics.* North-Holland, 1984.
3. L. Henrio and F. Kammüller. A Mechanized Model of the Theory of Objects. Accepted at *9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS 2007.* Vol. 4468, LNCS, Springer, 2007.
4. S. Jähnichen and F. Kammüller. *Ascot: Formal, mechanical foundation of aspect-oriented and collaboration-based languages.* Web-page at `http://swt.cs.tu-berlin.de/~flokam/ascot/index.html` Project with the German Research Foundation (DFG), 2006.
5. F. Kammüller. *Exploring New OO-Paradigms in HOL: Aspects and Collaborations. Theorem Proving for Higher Order Logics, TPHOLs'05*, 2005.
6. F. Kammüller. *Interactive Theorem Proving in Software Engineering.* Habilitationsschrift (habilitation thesis), Technische Universität Berlin, 2006.
7. F. Kammüller and H. Sudhof. *A Mechanized Framework for Aspects in Isabelle/HOL.* 2nd Informal ACM SIGPLAN Workshop on Mechanizing Metatheory, Universität Freiburg, 2007.
8. F. Kammüller and H. Sudhof. Composing Safely — A Type System for Aspects. *7th International Symposium on Software Composition, SC'08.* Satellite to ETAPS'08. Vol 4954 LNCS Springer, 2008.
9. F. Kammüller and M. Vösgen. Towards Type Safety of Aspect-Oriented Languages. In *Foundations of Aspect-Oriented Languages*, AOSD'06, 2006.
10. J. Ligatti, D. Walker, and S. Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming: Special Issue on Foundations of Aspect-Oriented Programming.* Springer 2006.
11. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, Springer LNCS, **2283**, 2002.
12. The POPLmark challenge. `http://alliance.seas.upenn.edu/ ~plclub/ cgi-bin/ poplmark`. July 2007.
13. C. Urban et al. Nominal Methods Group. Project funded by the German Research Foundation (DFG) within the Emmy-Noether Programme, 2006.