

# Towards Type Safety of Aspect-Oriented Languages

Florian Kammüller  
flokam@cs.tu-berlin.de

Matthias Vösgen  
mvoesgen@cs.tu-berlin.de

Technische Universität Berlin  
Fakultät IV - Elektrotechnik und Informatik  
Institut für Softwaretechnik und Theoretische Informatik  
Fachgebiet Softwaretechnik  
Sekt. FR 5-6, Franklinstr. 28/29, D-10587 Berlin

## ABSTRACT

In this paper we describe our vision of supporting aspect-oriented programming with interactive theorem proving in Coq. The goal is to construct a mechanized semantical basis that enables the analysis of type safety and other security properties. This basis consists of a featherweight aspect-oriented language and a supporting framework for analysis. We identify features of aspect-oriented programming that direct our investigation and present first axiomatic formalizations building the basis for this analysis of aspect-oriented programming.

## 1. INTRODUCTION

Computer aided reasoning (CAR) tools like Isabelle or Coq [2] have emerged as powerful tools for proofs too tedious for the human mind. Proofs of type safety of the language Java [10] and its JVM [9] have been a major application of CAR.

There has already been considerable research on formal foundations for aspect-oriented programming (AOP) languages, for example [12]. Ideas to redefine AOP mechanisms to make AOP safer have also been presented [5]. However, there appear to be few attempts of mechanized formalizations to investigate AOP.

Our plan is to use the interactive theorem prover Coq to put the concepts of AOP on solid ground. This would enable us to state and prove properties over these concepts like *“The default AspectJ aspects cannot violate confinement. Privileged aspects are able to violate the confinement rules.”* Moreover, we intend to build a framework that enables the stepwise construction of a theoretically sound and mechanically proved featherweight AOP language.

The aim is not only to have such a language but also to use the means of the framework to identify the effects on the safety and security properties of the resulting language if

specific aspect constructs are added. In formalizing the core concepts of AOP we rely on Featherweight Java (FJ) as a representative Java-like base language.

The view we take on AOP is somewhere in between an execution model, like [6], and a static language based one. Although we endeavour into the conception of a small representative AOP language by formalizing and analyzing its syntax and semantics, we consider the dynamic nature of AOP by considering, what we call, run-time weaving. That is, we design our framework such that it postulates that weaving has to respect compilation to bytecode.

In the next section we introduce very briefly the system Coq – concrete syntax is going to be explained in places where it is used. We present an outline of the formalization of a featherweight aspect language and some early type safety results in Section 2. Then we resume the runtime weaving problem in Section 3. Finally Section 4 contains our conclusions and future plans.

### 1.1 Coq

Coq [2] is an interactive proof assistant based on constructive type theory. Hence it is possible to generate executable Caml code from proofs. The programs are extracted to OCaml, an object-oriented extension of ML that is also the implementation language of the Coq system. A development in Coq consists of type and term declarations modelling a given application logically. This is a process of formal specification quite useful for computer science applications as the types enable to encode quite naturally features of the domain. Also it is possible to define functions in a style quite like a functional programming language. An excellent introduction to working with Coq is [1].

A very well-suited device in Coq to specify data structures is the inductive definition. It facilitates specification and proof because the prover automatically provides induction and inversion rules. In Coq functions can be defined similar to ML or any other functional programming language. Using the `Fixpoint` construct even recursive functions may be formalized. We are going to explain these features by means of example in the following section where we introduce Featherweight Java in Coq.

## 1.2 Featherweight Java in Coq

Featherweight Java (FJ) [7] has been devised as a representative subset of Java that is well-suited for the analysis of type safety of Java and related languages. It differs slightly from the original language in that methods resemble functions in functional programming languages. Otherwise it keeps to the concepts of the classic Java language.

This section briefly introduces some recent work on formalizing FJ in Coq [13] thereby also illustrating the use of the basic Coq features we use in our formalization.

A good example for an inductive definition is the formalization of FJ's method expressions `exp`.

```
Inductive exp : Set :=
| Var : varName → exp
| FieldProj : exp → fieldName → exp
| MethodInvk : exp → methodName → list exp → exp
| New : className → list exp → exp
| Cast : className → exp → exp.
```

An expression can be a variable, a projection to a field, a method invocation, a new construct, or a cast. Some expressions can contain nested expressions. Because of this it is necessary to use an inductive definition here.

Functions can now be defined over inductive structures in Coq as illustrated in the following function `subst` that formalizes substitution in the above expression type `exp`.

```
Fixpoint subst (sigma : (varName → exp)) e {struct e}
: exp :=
match e with
| Var x ⇒ sigma x
| FieldProj e fn ⇒ FieldProj (subst sigma e) fn
| MethodInvk e mn e1 ⇒
  MethodInvk (subst sigma e)
  mn (map (subst sigma) e1)
| New C le ⇒ New C (map (subst sigma) le)
| Cast C e0 ⇒ Cast C (subst sigma e0)
end.
```

This function substitutes variable names with the expression they represent by applying itself to every subexpression of complex expressions and replacing names of variables with an expression defined by a function `sigma`.

The use of Coq's module system facilitates the kind of axiomatic framework construction we are aiming at. Similar to ML structures, signatures, and functors, Coq offers modules to structure a formalization [3]. Thereby, local assumptions can be bundled in interface signatures and mechanizations can be reused.

## 2. FORMALIZING AOP

In the following we will give an outline of our formalization of AOP based on Featherweight Java and present the type safety problem. We show up where type safety in this naïve view of AOP goes wrong and identify the constraints that we are going to use to tackle the problem of type safety for AOP.

When we speak of the AO concepts in the following we use the common terms branded by AspectJ. So we speak of aspects, pieces of advice and pointcuts.

### 2.1 Formalization of Core AO Concepts

In a top-down way of approaching the problem of defining a formal model for AOP we begin by extending the base language FJ with aspects. We introduce pointcuts, and advice and define weaving over our program representation.

#### 2.1.1 Aspects

Aspects are like classes containers for methods and fields. Additionally they contain advice and pointcuts. Like the methods of classes in [13] our pointcuts and pieces of advice are stored in a pointcut table and an advice table respectively. We ignore other issues like priority annotations.

```
Inductive aspectDef : Set :=
  Aspect : aspectName → aspectName → list fieldDef
  → methodTable → pointcutTable
  → adviceTable → aspectDef.
```

An aspect is defined by a name, a name of a super aspect, fields, a method table, a pointcut table, and an advice table.

#### 2.1.2 Advice

Advice in AspectJ is declarable as before, after or around advice. We decided to constrain ourselves to around advice. It is possible to simulate before and after advice using around advice.

Because of the functional nature of FJ we had to think about the semantics of around advice. In a functional language “before” and “after” have no meaning. It is possible, however, to understand a surrounding functional expression as being “around” an inner expression.

Following this train of thought we come to an interpretation of around advice for a quasi functional language like FJ: the expression of an around advice is a well-formed expression of the language that can additionally contain a proceed statement at certain points. Weaving for a method means to replace the original method with a new one consisting of the advice expression in which every occurrence of proceed is replaced by the expression of the original method. The adapted method takes the same parameters as the original method. The parameters in the advice are the parameters of the adapted methods.

Thus no parameter passing from the advice to the adapted method takes place in our formalization. Using this simplified approach, `proceed` does not take any arguments and we don't have to decide if the number and the types of those arguments must match with the arguments of the advice like in AspectJ or with the arguments of the adapted methods like in the formalization `MiniMAO1` [4].

```
Inductive adviceExp : Set :=
| proceed : adviceExp
| adVar : varName → adviceExp
| adFieldProj : adviceExp → fieldName → adviceExp
```

```

| adMethodInvk : adviceExp → methodName
  → list adviceExp → adviceExp
| adNew : className → list adviceExp → adviceExp
| adCast : className → adviceExp → adviceExp.

```

```

Inductive adviceDef: Set :=
| aroundAdvice: pointcutName → adviceExp → adviceDef.

```

An advice expression is formalized similar to the expression type of FJ but introducing the constant advice expression `proceed`.

The advice is defined by the name of a pointcut and an advice expression.

### 2.1.3 Pointcut

A pointcut can be understood as a selection of program points. There are various possibilities to define a selection. Some of these are as follows.

- simple syntactic selections like listing a number of methods.
- syntactic selections that need an analysis of the syntactic structure like selecting all invocations of methods of subclasses of a class.
- lexical selections like selecting methods the names of which match a pattern.
- semantic selections like selecting method invocations that occur in a certain control flow.

We formalized the simple syntactic selection of methods by explicitly listing the methods.

```

Inductive pointcutSelection : Set :=
| methodSel: className → methodName
  → pointcutSelection.

```

```

Inductive pointcutDef: Set :=
| Execution : pointcutName → (list pointcutSelection)
  → pointcutDef.

```

A pointcut selection is defined by a class name and a method name. The pointcut itself has a name and lists selections. Because of its resemblance to the AspectJ execution pointcut we called this kind of pointcut execution pointcut. The advantage of a formalization in a theorem prover like Coq is that we can easily lift this simple list selection to general predicates. Using a suitably defined filter function we embed arbitrarily selected pairs of `pointcutSelection` into an `Execution` pointcut.

```

Definition ptct (CT: className)
  (pred: className → methodName → bool)
  (pn: pointcutName): pointcutDef :=
let pcl := filter pred CT
in Execution pn pcl.

```

It is then just the predicate that has to be specified to integrate different forms of advice. So, we believe our approach is likely to scale up although we restrict our attention here to the execution pointcut.

### 2.1.4 Weaving

Weaving means that every advice in every aspect of the aspect table adapts the methods selected by its pointcut.

The function `wv_AT_CT` (weave advice table into class table) is defined as follows:

```

Definition wv_AT_CT (CT: classTable) (AT: aspectTable) :
classTable :=
MapCollect _ _ (fun _ asp ⇒ wv_asp_CT CT asp) AT.

```

This function iterates over the aspect table and then for every aspect over its advice table. For brevity, we omit the hierarchies traversed from there and go directly to the definition of `merge_expr`. It adapts a method expression `mExpr` by merging it with the advice expression `aExpr`.

It does so by parsing the advice expression and replacing every occurrence of `proceed` with `mExpr` and every other advice expression with an equivalent method expression.

```

Fixpoint merge_expr (mExpr: exp)
  (aExpr: adviceExp) struct aExpr
: exp :=
match aExpr with
| proceed
  ⇒ mExpr
| adVar v
  ⇒ Var v
| adFieldProj aExpr2 fieldN
  ⇒ FieldProj (merge_expr mExpr aExpr2) fieldN
| adMethodInvk aExpr2 methodN aExprList
  ⇒ MethodInvk (merge_expr mExpr aExpr2)
    methodN (map (merge_expr mExpr) aExprList)
| adNew classN aExprList
  ⇒ New classN (map (merge_expr mExpr) aExprList)
| adCast classN aExpr2
  ⇒ Cast classN (merge_expr mExpr aExpr2)
end.

```

## 2.2 Type Safety

A generally accepted definition of type safety due to Pierce [11] is that well-typed programs do not go wrong, i.e. a well-typed term is not stuck: it can take a further step according to the reduction rules of the semantics or it is a *value* (*progress*), and if a well-typed term takes a step of evaluation then the resulting term is also well-typed (*preservation*).

The first step to prove the security of a programming language is to ensure that it is type safe. The property that defines well-typedness of an aspect-oriented program is given in the following axiom.

```

Axiom type_soundness_woven:
∀ (AT:aspectTable) (CT:classTable),
  class_table_typing CT
  → adv_table_typing AT
  → class_table_typing (wv_AT_CT CT AT).

```

Based on this property we can now prove the classical type safety property for aspect-oriented programs: a well-typed aspect-oriented program cannot go wrong.

```

Lemma weave_type_safety:

```

```

∀ (CTO CT: classTable)(e e': exp)(AT: aspectTable),
  CT = wv_AT_CT CTO AT
  → class_table_typing CTO
  → adv_table_typing AT
  → multi_step CT e e'
  → forall C, typing CT emptyCtx e C
  → ¬(∃ e'', reduction CT e' e'')
  → (value e' ∨ failed_cast CT e').

```

However, the property `type_soundness_woven` is not generally true. This is not surprising as an advice can be very liberally chosen and integrated at basically any point. For example, we can construct an advice that just evaluates the selected method and passes the evaluated value to a function that takes an integer and returns it unchanged. If this advice is applied to a method whose return type is string, then the result is obviously a call to a method that expects an int and gets a string.

We formalized this counterexample in Coq and got a type-unsound class table as a result (see Appendix).

From this counter-example we see that the critical point where the type safety is violated lies in the selection of the pointcuts. This is a first sign that our general strategy works: from our thorough mechanical formalization we arrive quickly at points where desired properties, here type safety, fail and can develop amendments. In our model it is easy to check if the types of the adapted method and those of the advice fit together. We can simply integrate it into the notion of aspect and aspect weaving.

### 3. THE RUNTIME WEAVING PROBLEM

This section summarizes our first axiomatic formalizations of aspect bytecode already presented in [8]. It resumes the major question of the paper and shows up how the formalizations presented in the previous section are going to be integrated with the formalization of bytecode.

When considering the process of weaving we have two ways to proceed:

- compile-time weaving: weave the advice in at the source code level, then compile,
- or run-time-weaving: translate program and advice (and join point selections) separately and weave in at the execution level.

We see immediately that run-time weaving is more desirable but clearly also trickier. Compile-time weaving is fairly simple.

For example, declarative programming languages like Prolog may be considered as aspect-oriented languages. Although there the distinction of compile-time does not apply, we can easily consider Prolog as a pointcut definition language for any imperative oo-language. To produce a compile-time weaver, just preprocess the source code using the Prolog predicates, then use the old compiler.

However, at times where global computing is on demand, run-time weaving is desirable because we want to be able to

adjust executable, deployed, program components by weaving in advice.

Based on the model of a compiler function and two weave functions for source and bytecode, we can now consider the property that represents the question whether run-time weaving is possible. More precisely, we try to identify the conditions such that the diagram in Figure 1 commutes.

$p$  stands for the an unadvised program,  $p'$  for an advised program,  $jp$  for a selection of join points and  $adv$  for a set of advice. The index `sc` stands for source code, the index `bc` for bytecode.

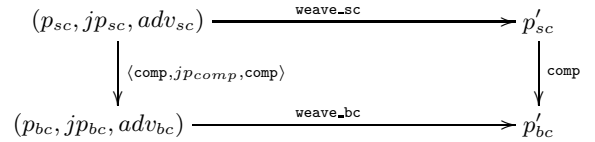


Figure 1: commutation of weaving

More technically, we want to show that the following is a theorem.

```

Axiom run_time_weave: ∀ p_sc: class_sc
  ∀ a_sc: advice_sc, ∀ mp: (method → Prop),
  comp (weave_sc p_sc (call_sc mp) a_sc) =
  weave_bc (comp p_sc)(call_bc mp)(comp a_sc).

```

The method predicate `mp` is the pointcut selector. It is the same for both levels as we assume that method names are not changed during compilation – which is true for Java. The constructors `call_sc` and `call_bc` filter out program points  $\mathcal{P}$  containing method invocations.

```

Definition call_bc : (method → Prop) → (P → Prop) :=
  fun (mp: method → Prop)(pc: P) =>
    match p_bc pc with invoke m => True
    | _ => False
  end.

```

To guarantee the commutation property `run_time_weave` we just have to construct a compiler function in which the weaving of advice into selected methods with `wv_mth` and the compiling of those methods commute.

```

comp_method(wv_mth_sc m adv) =
wv_mth_bc(comp_method m)(comp adv).

```

The goal is to arrive at an axiomatic framework in Coq that enables the check of the commutation property for various different pointcut selectors of aspect-oriented languages. The `call` selector is simple from a theoretical point of view as it enables selection purely based on syntax. For the `cfow` pointcut selector that uses predicates over the control flow of the program the commutation property will be more complicated: we need to use semantical equivalence rather than syntactical equality.

## 4. CONCLUSIONS AND OUTLOOK

We presented an aspect-oriented extension for a Coq formalization of Featherweight Java. The pointcut language of the extension is able to adapt a set of methods by explicitly selecting them. We sketch how this scales up to more liberally defined selection predicates. We outlined the type safety proof for aspects and reduced it to a well-typedness condition that has to be met by an AOP language in order to be safe. We showed why a naïve approach to AOP fails to guarantee type safety and offer an amendment.

We plan to extend our approach by successively adding pointcut constructs. Although we have for now only instantiated our advice formalization for execute pointcuts, we believe our approach to scale up.

Finally, we will continue to extend the framework such that we can address runtime weaving issues. That is, we construct a compiler function that corresponds to the formal requirement of respecting the commutation property of weaving. As we are defining a sound core language we are at ease to define a compile function in Coq and observe precisely which are the extensions to the aspect-definition devices that destroy the commutation.

## 5. REFERENCES

- [1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq’art: the Calculus of Inductive Constructions*. Springer-Verlag 2004.
- [2] Coq Development Team. *The Coq Proof Assistant User’s Guide. Version 8.0*, January 2004.
- [3] J. Chrzaszcz. *Implementing Modules in the Coq System*. In Proc. TPHOLs’03, LNCS volume 2758, pages 270 – 286. Springer-Verlag, 2003.
- [4] C. C. Clifton and G. T. Leavens. *MiniMAO: Investigation the Semantics of Proceed*. FOAL’05, Foundations of Aspect-Oriented Languages, 2005
- [5] Daniel S. Dantas and David Walker. *Harmless Advice*. Foundations of Object-Oriented Languages, FOOL’05, ACM, 2005.
- [6] R. Douence, O. Motelet, and M. Südholt. *A Formal Definition of Crosscuts*. In Reflection 2001, volume 2192 of LNCS, Springer, 2001.
- [7] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java — A Minimal Calculus for Java and GJ. In *Proceedings of OOPSLA’99, ACM SIGPLAN*, 1999.
- [8] F. Kammüller. *Exploring New OO-Paradigms in HOL: Aspects and Collaborations*. In TPHOLs 2005, Emerging Trends. Oxford, August 2005.
- [9] G. Klein and T. Nipkow. *Verified Bytecode Verifiers*. Theoretical Computer Science, 298(3):583–626, April 2002.
- [10] D. v. Oheimb and T. Nipkow. *Machine-Checking the Java Language Specification: Proving Type Safety*. In Jim Alves-Foss (Ed.): Formal Syntax and Semantics of Java, LNCS 1523, pp. 119–156, Springer-Verlag, 1999.
- [11] B. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
- [12] D. Walker, S. Zdancewic, and J. Ligatti. *A Theory of Aspects*. In ACM SIGPLAN International Conference on Functional Programming, 2003.
- [13] S. Weirich. *Type Safety of Featherweight Java in Coq*. <http://fling-l.seas.upenn.edu/plclub/cgi-bin/poplmark>, 2005.

## APPENDIX

Example for a type unsound aspect definition

```
Definition String := ad_x 1.
Definition StringDef: classDef :=
  Class String Object nil Constructor (M0 _).
Definition Int := ad_x 2.
Definition IntDef: classDef :=
  Class Int Object nil Constructor (M0 _).
Definition Example : className := ad_x 3.
Definition strFld := 1.

Definition ExampleFields : list fieldDef :=
  cons (String, strFld) nil.

Definition constructor := Constructor.
Definition getStr := ad_x 1.
Definition getStrExpr : exp :=
  FieldProj (Var this) strFld.
Definition getStrDef : methodDef :=
  Method String getStr nil getStrExpr.
Definition intIdent := ad_x 2.
Definition intParam := 1.
Definition intIdentExpr: exp:= Var intParam.
Definition intIdentDef: methodDef :=
  Method Int intIdent ((Int, intParam)::nil) intIdentExpr.

Definition ExampleMT : methodTable :=
  MapPut _ (M1 _ getStr getStrDef) intIdent intIdentDef.

Definition ExampleDef: classDef :=
  Class Example Object ExampleFields constructor ExampleMT.

Definition ExampleCT :=
  MapPut _ (MapPut _
    (M1 _ Int IntDef) String StringDef) Example ExampleDef.

Definition getStrPct := ad_x 1.
Definition getStrSel := methodSel Example getStr.
Definition getStrSelList : pointcutSelectionList :=
  getStrSel::nil.
Definition getStrPctDef : pointcutDef :=
  Execution getStrPct getStrSelList.

Definition violAdvExpr :=
  adMethodInvk (adVar this) intIdent (proceed::nil).
Definition violAdvDef :=
  aroundAdvice getStrPct violAdvExpr.

Definition ViolAsp := ad_x 1.
Definition ViolAspDef :=
  Aspect ViolAsp topAspect nil (M0 _)
  (M1 _ getStrPct getStrPctDef ) (M1 _ (ad_x 1) violAdvDef).

Definition ViolAT := M1 _ ViolAsp ViolAspDef.

Definition violatedCT := wv_AT_CT ExampleCT ViolAT.
```