

Exploring New OO-Paradigms with HOL: Aspects and Collaborations

Florian Kammüller

Technische Universität Berlin
Institut für Softwaretechnik und Theoretische Informatik

Abstract. In this paper we report on previous, current and ongoing work on the analysis of collaboration-based and aspect-oriented programming languages with mechanizations in Isabelle/HOL and Coq.

1 Introduction

Triggered by the ever increasing ubiquity of software the demand for more flexibility in the assembly of programming components is being felt more and more. Programs shall be run on small devices like handhelds, on dedicated operating systems, for example JavaCard for SmartCards. Moreover, application must be loadable on demand as resources in small devices are restricted. The concept of object is too fine grained to encapsulate entire applications — the obvious concept therefore is a module, or component as it was initially called. However, it turns out that the assembly of components is still somewhat too restricted. Components have no state, need the additional concept of deployment in different contexts and hence differ in instantiations which blurs the initially clean concept by creating multiple identities. Hence, a migration of already deployed components over the limits of execution environments is difficult.

Collaboration-based programming languages introduce a concept for modules for classes on top of the usual object-oriented language features. The basic idea is similar to packages, or components, but goes beyond that in that these modules can be instantiated. Thereby, we can define whole groups of interacting objects on the abstract, the class level. The defined groups can then be instantiated as a whole to build groups of interacting objects.

Consequently, there was a need to devise more flexible notions one structural level above object and class that would enhance the aforementioned mechanisms of dynamic loading and grouping. The new paradigms for object orientation that have been designed for this purpose are called mixins, traits or object teams. We call them unifyingly collaborations.

Besides this concept, that seems to be ideal for the development of large systems, there is the concept of aspect-orientation that introduces new concepts into object-orientation. Aspect-orientation enables the definition of small pieces of code, so-called aspects, that can then be distributed at certain points throughout an existing application. For example, a login could for security reasons be extended by a password check. Assuming the login is a feature that is used in

various parts of the code, one could now weave this aspect of password check into that code before each code segment that contains the login. Aspect-oriented languages comprise features for defining the aspects themselves and features for the definition of the points at which the aspects have to be woven into the code, so called join-points. An example for an aspect-oriented language is AspectJ [1], an extension of Java by aspect-oriented features.

A collaboration may as well be used as a wrapper, and a wrapper realizes an adaptation of behaviour, which is in turn nothing else than applying an aspect. The borderline between the concepts aspects and collaboration seems to be a bit vague.

Therefore, we think that a combination of collaborations and aspect-orientation is a promising direction for programming languages. In order to guarantee that such combined languages catch on, it is necessary to support them with the necessary theoretical foundation. Similar to the efforts to give formal models for Java that can be mechanically verified, we are working on the development of a formal, mechanically supported framework for collaboration-based, aspect-oriented languages in Higher Order Logic.

A formalization of the collaboration-based language Object-Teams in Isabelle/HOL, is a first example of exploring collaborations in HOL. It aimed at proving type-safety, and succeeded in proving confinement. It followed quite strongly the outline of the formalization of Java [11]. It turned out to be very complex, obscuring the most essential language features by being so close to an actual instance of a language with a rich set of features.

Therefore, we continued our work on a more general level. When considering a formal analysis of aspect-orientation we decided to work independently of a specific language, being just inspired by features as they are common in aspect-oriented languages, most prominently AspectJ – but not being restricted in our progress by specific language design decisions. We mechanized this model in Coq [3]. The reasons for this switch are that, first we could base the formalization of aspects on a fairly well-established model of bytecode [2]; second we are planning to do the type safety analysis by translating into League’s [10] intermediate language that is also mechanized in Coq.

We give an outline of this ongoing research in the paper: After a brief introduction into the ideas of collaborations and aspects in the remainder of this section, we summarize in Section 2 the formalization that has been done in Isabelle/HOL. We then present the work on aspects in Section 3. Finally in Section 4 we give an account of related work and our future plans.

1.1 Aspects and Collaborations

In this section we give a brief account on the principle concepts of aspect-oriented and collaboration-based programming languages.

Aspect-Orientation

The main idea of aspect-orientation is to adjust programs after their creation by *weaving* in at certain points (*point cuts*) some more program code (*advice*). The procedure of weaving is illustrated in Figure 1. The main concepts of aspect-

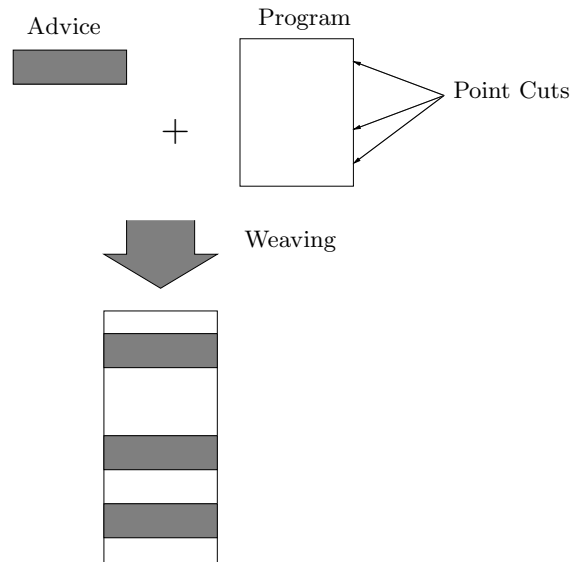


Fig. 1. Weaving advice at point-cuts

orientation, i.e. advices, weaving and join-points and the related point-cuts, are summarized as follows.

- advice is the code that has to be woven into the original program
- point cuts are the points in the program where aspects are woven in
- join-points are sets of point cuts usually described by some predicates using special keys like `call` to refer to all points of method invocation, and usual logical connectives.
- aspect-oriented language: the basic language is usually an object-oriented language, to express programs and advices
- join-point definition language: the language to express logical operators, quantification over program points.

Aspect-oriented program development usually starts by identifying so-called cross-cutting concerns. Starting from a usual object-oriented implementation in a standard way, the cross cutting concerns are then woven into it as advices.

Aspects: Observations 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-points) 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, for example Prolog, may be considered as aspect-oriented language. Although there the distinction of compile-time does not apply, we can easily consider Prolog as a join point 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.

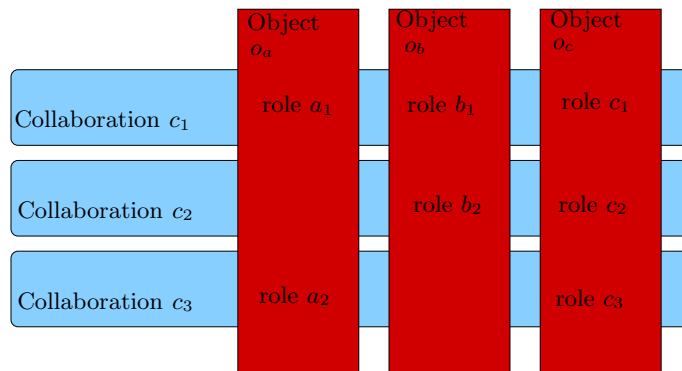
However, at times where global computing is on demand, run-time weaving is needed because we want to be able to adjust executable, deployed, program components by weaving in advice.

Collaboration-Based Programming

The main idea of collaborations is to consider collaborations of objects on the class level. Thereby, different from components, we can consider instantiation — like for usual classes — also for classes of classes, or collaborations. Conceptually, this feature enables the use of the collaboration idea at the source-code level, because classes correspond roughly to types. Therefore, using the collaboration concept we can statically check relationships between objects and can thereby express that

- Objects can be members of different collaborations
- Objects play *roles* in collaborations

If we furthermore adopt the ideas of inheritance for collaborations as well, we can even adapt existing object collaborations statically. To give a simpler



intuition that is independent of the concepts of object-oriented languages. The

typical introductory collaboration example is that of hotel, husband, wife, and daughter. If objects of these classes would want to collect at reception the keys for their family rooms in the hotel, a simple identification of the kind “I’m a daughter” does not suffice. Even being member of a super-class family-member does not suffice. Generally, a member object of a *group* needs to know to which instance of the group it belongs. As collaborations are considered on the class-level, i.e. the type-level, this membership creates a type dependency, i.e. the type of the group depends on instances of its constituting member classes.

2 ObjectTeams in Isabelle/HOL

The programming model of Object Teams [7] is an example of a collaboration-based language. The developers of Object Teams are currently working on extending the language to integrate aspect-orientation. For a theoretical idea on how this may be achieved see Section 4. The current version of Object Teams is called ObjectTeams/Java, because it uses Java as a host language, i.e. is translated into Java. But, in principle, any other class based-language could be used as host language. This language is well supported and is already used in practical applications in projects with industrial collaboration [6]. The module concept is called team, which is a container for so-called roles. The containment is on the level of classes and instances. Besides the classical inheritance that exists between classes the introduction of the concept of teams enforces a second implicit inheritance, as roles contained in teams inherit from each other, when a team inherits from another team. Teams thereby realize the concept of virtual classes with overriding. Virtual classes enable the refinement of sets of mutually-recursive types. Resulting issues of static type checking are handled by instance based types, a special kind of dependent types. By the combination of implicit inheritance and instance based types, teams realize the concept of family polymorphism [4]. On top of the aforementioned concepts, ObjectTeams/Java allows for different levels of role confinement providing for an ownership-like alias control [16]. Except for some flavors of role confinement these concepts have been formalized and the results of their analysis will be presented next.

2.1 Isabelle/HOL model of ObjectTeams/Java

The master’s thesis [14] provides a formalization of the main concepts of the language ObjectTeams/Java. This extensive experiment is strongly based on the works of Nipkow, von Oheimb et al [13, 11]. However, as we introduce completely new concepts, we had to redefine and reprove basically everything. We define abstract syntax and type system. The inheritance relation with its particular extension of implicit inheritance is defined. Well-structuredness of classes and the inheritance relation is proved. The type model is divided into static and dynamic instance based types. As role classes are contained in team classes, their instances have the same containment. Hence a roles type does depend on its instance context. That is what we call instance-based typing. In our model

we do not use dependent types explicitly, rather model them by extending the types in the static part by a fixed constant `TThis` that is then replaced in the dynamic typing by the actual instance context. Thereby we can separate the analysis into static and dynamic part. In order to define well-formedness predicates for the type-safety we define typing rules using inductive definitions. An operational semantics is modelled using inductive definitions as well. Finally, we have partly proved the type soundness theorem. However, we did not succeed as the formal analysis already produced a fundamental problem of the language: as `ObjectTeams` allow the instantiation of abstract role classes, type checking a sub team class requires more information than type checking a conventional subclass. Besides a super team class signature, a list of so-called relevant role classes is necessary for the judgment of well-formedness of the sub team class. Relevant role classes are those role classes that a sub team must implement in order to be concrete. This feature has been discovered during the formalization work and is now integrated into the language. However, it prevented us from completing the mechanical proof of type soundness. Assuming for the time being the type soundness as axiom we could, however, prove the following confinement statement.

```
[| G |- (x,(h,l)) - t >-> (v,x,(h,l));
  wf_prog G; conf (h,l) (G,L); (G,L) |- t :: T |]
==> (x'=None -> role_referenced_in_context G (h,l) v)
```

The theorem expresses that role objects are only referenced within the context of their enclosing team instance. The proviso has to be read as: a term `t` evaluates with respect to program `G`, producing a value `v`, transforming the state of heap and local environment `(h,l)` into `(h',l')`. Furthermore, it is assumed that the program is wellformed, that the before-state `(h,l)` conforms to the static environment `(G,L)`, that the term `t` is well-typed, and that no exception occurred `x'=None`.

3 Coq Aspects

In this section we present the mechanization of aspects in Coq. The main goal of this formalization is to supply an axiomatic framework that helps to identify conditions for aspects oriented languages to work. The major interest is to realize run-time weaving. That is, it should be able to weave in advice at run time.

3.1 Basis: OO-language

We use a formalization of the main concepts of object-orientation given in [2] – an axiomatic framework for non-interference. Although originally designed for proving and deriving a bytecode-verifier it does also provide a mechanization of the operational semantics of the Java bytecode language. Although we want to treat aspects and collaborations on an abstract level, we think that using the Java bytecode as the execution layer is not such a strong restriction of

generality because (a) our model treats basic features that are quite common for most object-oriented languages (b) most object-oriented languages can be translated (and are in fact translated) to the Java bytecode in order to gain wider applicability. For the latter point even C++ offers a translation to Java bytecode.

We give a very short outline of the relevant properties of this basic Coq model leaving out the details that refer to security problems. The bytecode instruction set we consider is formalized using the following inductive definition.

```
Inductive instr : Set :=
  nop : instr
| push : value -> instr
| iadd : instrin
| load : locs -> instr
| store: locs -> instr
| goto : ppt -> instr
| ifthenelse : ppt -> instr
| new: Class -> instr
| getfield: Field -> instr
| putfield: Field -> instr
| invoke: method -> instr
| retrn: instr.
```

The set `ppt` is a decidable set of program points and `locs` are the register locations. We assume a program `p` to be a map from program points and method names to instructions. Based on this instruction set the operational semantics is defined introducing operand stacks, object heaps, values as structured types containing pointers and simple values, and register valuations. A state in the semantics is constituted by a stack of method frames. Each of those frames comprises the current program counter, a current variable binding, an operand stack, and a security environment. Both the variable binding and the security environment are treated abstractly with lookup and update functions.

```
Record frame: Set:= {pc: pcs; rm: env ; os: stack}.
```

States are composed as stacks of frames and a heap.

```
Record state: Set:= {fs: stack(method × frame); hp: heap }.
```

In the operational semantics we define the execution of a program step-by-step over program states by a simple case analysis over the type of instructions and assigning the corresponding effect on the state. A general n -step execution `exec` is then defined inductively over this one step execution function.

Now in order to be able to reason about source code we have to build a second instruction layer of source code instructions.

```
Inductive sc_instr: Set :=
  assign : var -> value -> sc_instr
| add : var -> var -> sc_instr
| ifte : (var -> bool) -> ppt -> ppt -> sc_instr
```

```

| New: Class -> sc_instr
| Getfield: Field -> sc_instr
| Putfield: Field -> sc_instr
| Invoke: method -> sc_instr
| Retrn: sc_instr.

```

and correspondingly develop the notion of state and execution similar to the bytecode level. Compilation between the two levels will be considered next, but first we model join-points.

3.2 Call Join-Points and Weaving

The most frequently used join-point constructor used in aspect-orientation is the `call` construct. With the `call` operator we can construct a predicate that selects program points that contain a method invocation. As arguments to the `call` operator we can, for example in AspectJ, use so-called *wildcards*, annotated by `*`, to create some kind of pattern matching over method names. In the formalization we can easily be more general than this by just allowing any predicate over method names, type `method`, as admissible input to the `call_sc` constructor for call join points at the source level.

```

Definition call_sc : (method -> Prop) -> join_point :=
  (fun mp: (method -> Prop) => fun pc: ppt =>
    match p_sc pc with Invoke m => True
    | _ => False
  end).

```

The type `join_point` is just an abbreviation for `ppt -> Prop`, i.e. predicates over program points.

Compilation of a source program replaces instructions by creating a new binding of program points to bytecode-level instruction. The call join-point constructor is translated as follows.

```

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

```

As weaving can be performed as an additional step of compilation, it can as well be seen just on the syntactic representation of programs. In order to analyze the requirements of a run-time weaving we start from assumptions about the compilation process from source to bytecode level.

Assuming a compilation function mapping source code programs `program_sc` to bytecode-programs `program_bc`, we can reason about the weaving process in order to gain precise information about the inherent requirements of aspects.

Parameter `comp: program_sc -> program_bc`.

We assume in a first step only non-optimizing compilers, i.e. we assume that each source code command is one to one translated into a sequence of bytecode-level instructions. That is, we assume that the compilation is injective. Hence, we can assume an inverse function.

Parameter `compinv`: `program_bc` \rightarrow `program_sc`.
 Axiom `compinj`: \forall `p_sc`: `program_sc`, `compinv`(`comp` `p_sc`) = `p_sc`.

Weaving simulation Based on the model of a compiler function, we can now consider the property that represents the question set out in the beginning (cf. 1) whether run-time weaving is possible. More precisely, we try to identify the conditions such that the diagram in Figure 2 commutes. The two weaving functions

$$\begin{array}{ccc}
 (p_{sc}, jp_{sc}, adv_{sc}) & \xrightarrow{\text{weave}_{sc}} & p'_{sc} \\
 \downarrow \langle \text{comp}, jp_{comp}, \text{comp} \rangle & & \downarrow \text{comp} \\
 (p_{bc}, jp_{bc}, adv_{bc}) & \xrightarrow{\text{texec}} & p'_{bc}
 \end{array}$$

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

are represented by the types

Parameter `wv_sc`: `program_sc` \rightarrow `join_point` \rightarrow `advice_sc` \rightarrow `program_sc`.
 Parameter `wv_bc`: `program_bc` \rightarrow `join_point` \rightarrow `advice_bc` \rightarrow `program_bc`.

In order to narrow down the possible specification of these functions we identified the following assumption.

Axiom `step1`: \forall (`pbc`: `program_bc`)(`jp`: `method` \rightarrow `Prop`)(`adbc`: `advice_bc`),
 \forall `wpbc`: `program_bc`,
`wpbc` = (`wv_bc` `pbc` (`call_bc` `jp`) `adbc`) \rightarrow
 (`exists` `p_sc`: `program_sc`,
`comp` `p_sc` = `wpbc` \wedge `p_sc` = `wv_sc` (`compinv` `pbc`)(`call_sc` `jp`)(`compinv` `adbc`)).

This property states that a bytecode-weave does only produce programs that can also be created as source-code. With this property we can prove the commutation property.

Lemma `run_time_weave`: \forall `mp`: (`method` \rightarrow `Prop`),
 \forall `a_sc`: `advice_sc`, \forall `a_bc`: `advice_bc`,
`comp` `a_sc` = `a_bc` \rightarrow
`comp` (`wv_sc` `p_sc` (`call_sc` `mp`) `a_sc`) =
`wv_bc` (`comp` `p_sc`)(`call_bc` `mp`)(`a_bc`).

Although we can under reasonable assumptions prove the correctness of run-time weaving here, this is only restricted to the standard `call` join-points.

3.3 Logical connectives and Control Flow

The definition of join points offers in general further possibilities beyond the `call` construct. Mixed join-point expressions can be introduced using logical connectives. The following rule is an instance of the major property shown in the previous section but for the disjunction of join-point expressions.

```
Axiom rt_disj: ∀ jp1 jp2: join_point,
  ∀ a_sc: advice_sc, forall a_bc: advice_bc,
  comp a_sc = a_bc ->
  comp (wv_sc p_sc (jp_disj jp1 jp2) a_sc) =
  wv_bc (comp p_sc)(jp_disj jp1 jp2) a_bc.
```

In order to deal with such combined weaving processes a decomposition may be achieved using the following lemma to iterate the construction.

```
∀ jp1 jp2: join_point,
  ∀ a_sc: advice_sc, forall a_bc: advice_bc,
  wv_sc p_sc (jp_disj jp1 jp2) a_sc = wv_sc (wv_sc p_sc jp1 a_sc) jp2 a_sc.
```

In the example shown we illustrate disjunction. The decomposition does not work like this for conjunction of join point expressions. Here, further thought is needed.

Another possibility for constructing join-point expressions is the selection of join-points by selecting a control flow. Here the `cflow(c1, mn)` enables to select program points from the beginning to the end of method execution of method `mn` of class `c1`. For the analysis of weaving advice according to join-points selected in this manner the contemplation at the execution level, i.e. according to program behaviour, seems more appropriate.

3.4 Weaving equivalence based on Behaviour

The properties related so far are all based on the idea to show the correctness of run-time weaving at the syntactic level. For example, using the execution function, provided in the operational semantics [2] we can prove,

```
Axiom rt_conform: ∀ jp1 jp2: join_point,
  ∀ a_sc: advice_sc, forall ,
  exec (comp (wv_sc p_sc (call_sc mp) a_sc) =
  exec (wv_bc (comp p_sc)(call_bc mp)(comp a_sc)).
```

by just substituting with lemma `run_time_weave`. Hence, the condition we have derived is clearly sufficient, for `call`, but probably a bit strong in general. Possibly, when translating onto different versions of bytecode, it might be that we cannot rely on the conformance relations between compilers and program points. Optimizing compilers could also destroy the injectivity assumptions.

Here, we have to show property `rt_conform` directly over the operational semantics not syntactical equivalences.

4 Related Work and Outlook

4.1 Intermediate Language LITL

As mentioned before League and Monnier devise an intermediate language called LITL that aims at type preserving compilation of class-based languages. LITL is based on the intermediate language Links [5]. Similar to this predecessor the intermediate language LITL enables to compile various flavours of class concepts, including collaborations (there called traits or mixins). Moreover, in contrast to Links, LITL is typed, as it is embedded into Coq. The typing of LITL is the major contribution of this work. Although the authors have already achieved a typing of LITL the second point – the translation of collaborations onto LITL, and thus the typing of collaborations — is not yet finished.

For our project LITL is a well-suited target language, as we can experiment with language features and — given the translation works — can see whether typability by typed compilation onto LITL is preserved. However, as LITL does not support aspects directly, we have to find a way to either integrate aspects into collaborations or find out what are the fundamental differences that could clarify the differences between the concepts. For the former possibility we consider a way to integrate aspects and collaborations.

4.2 Aspects as Collaborations

The question that we address in this subsection is what have aspects to do with collaborations. We present the conceptual idea how to use collaborations to represent aspects. This method can in principle be used in our mechanization to create a unified framework for aspects and collaborations. As we see in Figure

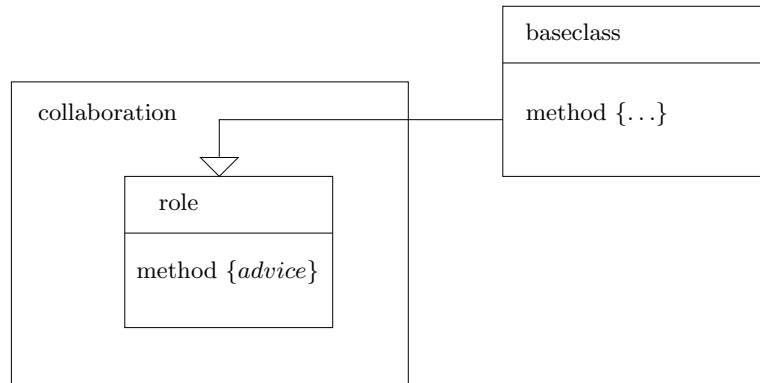


Fig. 3. Aspects represented by collaborations

3 collaborations act as wrappers. Using the possibility of method redefinition in

a collaboration the role class adjusts method `method`. That is, we realize aspects by a navigation between role classes and their base classes. Instead of using method over-riding as in the example, we could use so-called callin and callout constructors, as they exist for example in Object-Teams, to adjust the base class method.

Although in principle, this mimics the behaviour of advice weaving, it leaves open how to define such adjustments of existing behaviour in a controlled way similar to the concepts of join-points and weaving. The development of join-point languages for collaborations, in order to specify and quantify the locations for the advice application is currently under research with the language developers. Nevertheless, we think about offering solutions on the level of mechanized specification.

4.3 Discussion

We have introduced the notions of collaborations and aspects and described in outline a mechanization of the collaboration concept in Isabelle/HOL as well as an axiomatic approach to aspects in Coq.

The mechanization of collaborations at the example of Object Teams showed up difficulties with respect to type safety and enabled the proof of a confinement property, given type safety. The abstract formalization of aspects yielded properties that are a suitable frame for the analysis of concrete weaving functions with respect to facilities for the definition of join points and the crucial question of run-time weaving.

The translation of the formalization of ObjectTeams into Coq will enable us to experiment with the mechanical analysis of the integration of Aspects and Collaborations as sketched in the previous section.

We aim at providing a mechanized logical framework that enables the experimentation with language features, like exception handling, dynamic class loading, method overriding, in order to see if typability and security features are violated. This is precisely the kind of tool that is needed for the support of the language development: a workbench that enables to test the implication of the redefinition of generalization of a language concept. The challenge for this project is whether we will – beyond analyzing the principal language properties of a restricted sublanguage – be able to answer this need.

References

1. ASpectJ – Java with Aspects. www.eclipse.org/aspectj.
2. G. Barthe and F. Kammüller. Certified Bytecode Verifier for Noninterference. Technical Report, INRIA Sophia-Antipolis, 2005. In print.
3. Coq Development Team. *The Coq Proof Assistant User's Guide. Version 8.0*, January 2004.
4. E. Ernst. Family polymorphism. In *Springer LNCS 2072*, 2001.
5. K. Fisher, J. Reppy, and J.G. Riecke. A calculus for compiling and linking classes. In *Programming Language Design and Implementation*, 2000.

6. S. Herrmann. TOPPrax - Aspektorientierte Programmierung für die Praxis. <http://www.topprax.de>.
7. S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Springer LNCS 2591*, 2003.
8. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java — A Minimal Calculus for Java and GJ. In *Proceedings of OOPSLA'99, ACM SIGPLAN*, 1999.
9. F. Kammüller. Modular structures as dependent types in isabelle. In *Springer LNCS 1657*, 1998.
10. C. League and S. Monnier. Typed Compilation Against Non-Manifest Base Classes. In CASSIS '05, March 2005, Nice. To appear in Springer LNCS.
11. T. Nipkow et al.. Java Source and Bytecode Formalizations in Isabelle: Bali. <http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/past.html>, 20.02.2004.
12. M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *Proc. ECOOP'03*, Springer LNCS, July 2003.
13. D. v. Oheimb. p *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.
14. R. Oeters. *Foundation of ObjectTeams/Java in Isabelle/HOL*. Diplomarbeit. Technische Universität Berlin, 2003.
15. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, Springer LNCS, **828**, 1994.
16. J. Vitek and B. Bokowski. Confined types in Java. *Software, Practice and Experience*, 31(6):507-532, 2001.