Essential AOP: The A Calculus

Bruno De Fraine[1], Erik Ernst[2,⋆], and Mario Südholt[3]

[1] Software Languages Lab, Vrije Universiteit Brussel, Belgium
bdefrain@vub.ac.be
[2] Department of Computer Science, Aarhus Universitet, Denmark
eernst@cs.au.dk
[3] Département Informatique, École des Mines de Nantes, France
sudholt@emn.fr

Abstract. Aspect-oriented programming (AOP) has produced interesting language designs, but also ad hoc semantics that needs clarification. We contribute to this clarification with a calculus that models essential AOP, both simpler and more general than existing formalizations. In AOP, *advice* may intercept method invocations, and *proceed* executes the suspended call. Proceed is an ad hoc mechanism, only usable inside advice bodies. Many pointcut mechanisms, e.g. wildcards, also lack regularity. We model proceed using first-class closures, and shift complexity from pointcuts to ordinary object-oriented code. Two well-known pointcut categories, *call* and *execution*, are commonly considered similar. We formally expose their differences, and resolve the associated soundness problem. Our calculus includes *type ranges*, an intuitive and concise alternative to explicit type variables that allows advice to be polymorphic over intercepted methods. We use calculus parameters to cover type safety for a wide design space of other features. Type soundness is verified in Coq.

1 Introduction

This paper proposes a model of aspect-oriented languages which separates essential aspect features from inessential ones, while being simpler, more regular and more expressive than existing models.

Starting with the ECOOP'97 keynote by Gregor Kiczales [1], aspect-oriented software development has been a very active community, experimenting with programming language design and implementation. The experiments generally aimed to improve support for separation of concerns using a restricted kind of compile time meta-programming, as witnessed by AspectJ [2], the currently predominant aspect model. The motivation was taken from real-world scenarios, including software development concerns such as logging and tracing, and the overall approach was dominated by experiments with full-fledged language implementations and enthusiastic support from a substantial community of users.

In the AspectJ model of AOP, *aspects* are class-like entities that may refer to and modify the execution of an application at *join points*, principled points

⋆ Supported by FTP 274-06-0029.

T. D'Hondt (Ed.): ECOOP 2010, LNCS 6183, pp. 101–125, 2010.

in the execution of a program. Aspects contain method-like *advice* declarations, which are used to intercept method invocations specified by declarative *pointcut* expressions, i.e., query expressions that denote sets of join points. Advice may be categorized as `before`, `after`, or `around`, which means that it will be executed before, after, or instead of the intercepted join point. In the latter case, the advice may invoke the intercepted join point using a mechanism known as *proceed*. New arguments to the join point may be provided in such a proceed invocation, and in case the join point is the call or execution of a method, the advice may also choose a new receiver; this is known as *receiver substitution*.

Despite the relatively unified underlying motivation, aspect-orientation has given rise to a plethora of concrete variants of language mechanisms with their associated runtime semantics and typing rules [3]. The result is that many interesting ideas are available, but the foundations have been somewhat unclear. For instance, even the most widely used and mature language, AspectJ, is still plagued by type soundness problems in the treatment of advice [4, Sec. 2].

One fundamental problem of mainstream aspect-oriented languages is that a number of features commonly considered essential for AOP languages have an ad hoc flavor. One of these features consists in the large set of very different pointcut mechanisms typically used to select execution events [5,6,7,8,9,10,11]. This is the case for most aspect languages, in particular AspectJ, but also for many formal models of AOP [12,13,14]. A precise general formulation of pointcuts in terms of more primitive concepts is, however, worthwhile to give a comprehensive account of mechanisms for join point selection. Similarly, the proceed mechanism is typically realized as a special method `proceed` that may be called within an advice body, but cannot be called elsewhere. Furthermore, receiver substitution as part of a call to `proceed` is typically governed by strong restrictions, *e.g.*, on the ordering of call and execution advice, that are not derived from first principles.

A second problem of AOP languages consists in a number of open issues concerning the type-safety of pointcut and advice, in particular, to provide integrated support for both *generic advice* (which augments the join point with additional behavior, but otherwise executes it unmodified), and *non-generic advice* (which replaces the join point with new behavior). Previous approaches such as MiniMAO_1 [14] and typed polymorphic aspects [15] have important restrictions on one of these two classes of advice behavior. StrongAspectJ [4] supports both classes, but uses separate mechanisms for each, which prevents a single advice from combining both classes of behavior. Moreover, type variables must always be explicitly declared in the generic case, which makes advice specifications more complex than those known from AspectJ.

We address the above problems through the A calculus, which is a simple calculus in the style of Featherweight Java [16] that clarifies the foundations of aspect-oriented languages. The A calculus models the essential features of AOP, *i.e.*, the ones that directly impact the dynamic semantics and typing of programs. Other, less essential features are not directly included—such as the specifics of particular pointcut mechanisms—but a wide design space for these features is

provided through the expressive power of the core framework. This results in a calculus that is simpler and more regular than existing calculi, yet supports a wider range of practical approaches. Compared to other aspect calculi the main concrete innovations of the *A* calculus are the following:

1. We model call and execution advice separately, since we show that their difference is so deep that they need to be treated differently in the type system. The separation is maintained for the proceed mechanism, which is modeled using two kinds of explicit, first-class closures. This positions proceed as an integrated part of the language rather than a special primitive. Overall, the resulting aspect typing provides a clean semantic account of the conditions necessary to use receiver substitution in an advice body.
2. We abstract over a large space of designs for advice selection by making the advice selection strategy an explicit parameter of the calculus. The *A* calculus is therefore a general framework for a range of aspect calculi, similar to HM(X) [17] which models a range of calculi whose type systems include constraints. We stipulate simple safety conditions on concrete advice selection mechanisms taking the place of this parameter. Based on these conditions we prove type soundness for all these advice selection mechanisms at once.
3. The support for different classes of advice behavior in the type system is improved over earlier work (in particular [4]). Both generic and non-generic advice behavior are supported in a unified typing construct that is straightforward to use, since type variables do not need to be declared explicitly.

The first and third features allow for a deeper understanding of central elements of aspect languages, such as call and execution advice, generic and non-generic advice behavior, and proceed. The second feature and (to a lesser extent) the first feature provide a wide design space for non-essential characteristics, such as different pointcut languages, support for aspect instances, and aspect inheritance. For example, the advice selection parameter may be used to express the static and dynamic pointcut designators of AspectJ, as well as many other proposals for pointcut languages; it also enables features such as dynamic aspect deployment, and redefinition of advice in a subclass (similar to virtual method redefinition). Moreover, even though call and execution advice methods[1] are explicitly separated, the use of first-class closures for the proceed mechanism allows us to implement advice behavior in ordinary methods that can be shared between different advice methods. In general, the ambition of the calculus is to reveal the essentials of AOP, while abstracting over non-essential features through abstract parameters that may be filled in at will, or through an encoding to standard object-oriented features. The calculus thus provides several means of abstraction that, albeit being different from the more ad hoc means traditionally used in aspect languages, allow a wide range of aspect-oriented concepts and mechanisms to be defined precisely.

[1] The noun 'advice' is singular; we use 'advice method' as a synonym, with the natural plural form 'advice methods'.

In addition to these contributions, the calculus and its type safety proof have been formalized using the Coq proof verifier[2], yielding a level of rigor that surpasses previous aspect calculi. The Coq specification contains novel techniques; in particular, the soundness proof is simplified due to the use of empty type environments.

Next, Section 2 gives an example-driven overview of the calculus. We present our calculus formally in Section 3. We state the safety properties in Section 4 and discuss the proofs that we have developed for these properties. Related work is discussed in Section 5. We conclude and present future work in Section 6.

2 Overview of the *A* Calculus by Example

This section presents the *A* calculus, its concepts and language mechanisms based on a number of examples. Along the way we demonstrate how the *A* calculus covers a larger part of well-known aspect-oriented features than is obvious from the syntax. In this section, we have extended the calculus with a number of pragmatic features, *e.g.*, the primitive type `int`, thus avoiding excessive verbosity in the examples. These extensions are listed in detail at the end of the section.

```
1  class C { int m1(int i, int j) { return i+j; }}
2  class D { void m2(int x, String s, int y) { System.out.println(x*y); }}
3  aspect A {
4    pointcut p(int a, int b):
5        execution(int C.m1(int,int)) && args(a,b) ||
6        execution(void D.m2(int,String,int)) && args(a,*,b);
7    Object around(int a, int b): p(a,b) { return proceed(a+1,b-1); }
8  }
```

Fig. 1. AspectJ example

```
1  class C { int m1(int i, int j) { return i+j; }}
2  class D { void m2(int x, String s, int y) { System.out.println(x*y); }}
3  class A {
4    int m((int,int)->int proceed, int a, int b) { return proceed(a+1,b-1); }
5    around1: execution(int C.m1(int a, int b)) { return m(proceed,a,b); }
6    around2: execution(void D.m2(int a, String s, int b)) {
7        m((int a, int b => proceed(a,s,b); return 0),a,b); }
8  }
```

Fig. 2. *A* example

First-class closures and `proceed`*.* Consider the AspectJ example in Fig. 1, which compiles without errors or warnings in AspectJ version 1.6.4. It demonstrates a few issues with the current mainstream approach to aspect typing. AspectJ typing will verify that the advice behavior is compatible with the join points to

[2] The Coq development is available at `http://soft.vub.ac.be/acalculus/`

which it is applied. For example, the advice return type should be assignable to the join point return type, such that the the advice method result may replace the join point result. However, the `Object` return type of the `around` advice (line 7) activates a very special kind of polymorphism which allows the actual intercepted method to have different return types, even `int` and `void` (for details, see [4, Sec. 2.3]). This is unsound: consider the case where the advice returns a general `Object` instance rather than the original return value from proceed, and the intercepted method has a non-`Object` result type. Furthermore, it is not obvious what the intended semantics should be in case the "returned `void` value" is used when advising method `m2`. In practice, the value is `null`.

Following earlier work, our approach applies a strict and sound typing discipline and hence avoids such problems. However, as we will discuss below, it is still able to exploit the reuse potential in the example, because the notion of `proceed` is first-class in the *A* calculus. Figure 2 shows a program in the *A* calculus which corresponds to the program in Fig. 1. The classes `C` and `D` and their methods `m1` and `m2` are unchanged. The aspect is now a class that contains a new method `m`, as well as two advice declarations (lines 5–7) whose syntax is a simplified variant of the syntax known from the AOP mainstream. We discuss the advice declarations first, and then return to the method `m`.

Advice declarations in the *A* calculus differ syntactically from the ones in AspectJ: they specify an advice name (such as `around1` and `around2`), but don't include a signature. Rather, the advice kind (call or execution), the type annotations, and the formal parameter names are specified at the place of the pointcut expression, in a form similar to a typical basic pointcut designator in AspectJ.

This 'pointcut designator' is used for selection of compatible advice at runtime, and for type checking. Advice compatibility is a very simple, purely type-based advice selection specification—which we consider as essential. The rest of the advice selection semantics is given as parameters of the *A* calculus. One parameter is an execution history datatype, which is used to collect information about the execution so far, and an initial execution history value. Another parameter is an execution history update function, which maps the execution history value to the next one at each step. The final parameter is an advice selection function that produces a list of compatible advice from several arguments, including the execution history. Concrete advice selection mechanisms may be expressed using these parameters, as long as they select advice compatible with the given join point (compatibility is defined formally in Section 3).

Note that the example includes method names (such as `m1` and `m2`) in the pointcut designators. The *A* calculus advice compatibility test actually ignores method names and only uses the type information, but the advice selection parameters can trivially be used to specify that the method names must match. Method name matching is assumed in this section as a pragmatic extension. Hence, the pointcut designators have the meaning that one would expect, based on the meaning of similar syntax in languages like AspectJ.

The pointcut expression in Fig. 1 uses a typical set of conjunctions and disjunctions, in that they match one or more specific method signatures and provide

access to the binding environment of each intercepted method call by means of a primitive pointcut designator, here `args`. In general, this kind of combination can be expressed in the *A* calculus by transforming each disjunctive branch of the pointcut expression into one advice declaration, resulting in the two declarations in lines 5–7 of Fig. 2. Since this transformation has replaced one advice by two, we need to extract the commonalities among them in order to preserve the level of abstraction and reuse. We do so by placing the common behavior in the method `m` on line 4. Since the intercepted join point (`proceed`) is invoked as a part of this common behavior, the ability to create this abstraction depends critically on closures being first-class in the *A* calculus, *i.e.*, on the ability of the method `m` to take a closure that represents `proceed` as an argument.

There are two kinds of closures in the *A* calculus: one that is invoked with a receiver as well as the method arguments (designated as *static* closures), and another (*dynamic*) one that embeds the receiver and only accepts method arguments. A dynamic closure is semantically similar to a C# delegate and a traditional closure, in that it encapsulates an environment (a receiver object) and a piece of code (a method from that receiver object); a static closure is similar to a C++ pointer to a virtual function member, in that it encapsulates a method identity and a choice of receiver type, and it involves method lookup when the actual receiver is provided and the method called. Both variants are needed in order to handle different kinds of advice correctly. In the example, the method `m` takes a dynamic closure as its first argument. Its type is a dynamic closure type that lists the types of the arguments (left of the arrow) and the type of the result (right of the arrow).

The only remaining issue is the transfer of context information between the advice and the method `m`. The situation is complicated due to the different argument lists of the two advised methods `m1` and `m2`. In AspectJ, the context information is transferred using `args` and similar primitive pointcuts, and the difference in arguments is handled by putting a '`*`' in the `args` pointcut, which will match one argument of any type. Additionally, '`..`' may be used to match a variable number of arguments, but this syntax may be used at most once per `args` pointcut, making this a somewhat ad hoc measure. Traditional pointcut language design has in fact produced a growing list of ad hoc mechanisms.

In the *A* calculus, the context information is made directly available through named argument lists in the pointcut designators. The advice methods `around1` and `around2` may then bridge the differences between the advised methods when they invoke the shared method `m`, eliminating the need for `args` and similar primitive pointcuts. Note that the difference between the advised methods implies that the `proceed` closures have different signatures. The first advice may use its closure directly in an invocation of the method `m`; for the second advice the types do not match, and hence we wrap the invocation of `m` in an anonymous closure.

We claim that the above two transformations (placing disjunctive pointcut branches in separate advice declarations, and routing context information) handle a very large part of the practical uses of logic connectives and primitive designators like `args` in pointcuts. The *A* calculus can thus be considered a

useful model of languages with elaborate pointcut mechanisms, even though its pointcuts appear very simple at first sight. Moreover, the advice selection parameters can express dynamic mechanisms such as `if` pointcuts and trace based pointcuts. In general, the *A* calculus obtains great expressive power through a few, carefully chosen concepts, by parameterization, and by shifting complexity to ordinary object-oriented abstractions.

```
1  class Component {
2    PaintImpl p;
3    void paint() { p.paint() };
4  }
5  class JTextArea extends Component { ... };
6  class JScrollPane extends Component { ... };
7
8  class Factory { Component create(String t) { return new JTextArea(t); }}
9
10 class A {
11   around1: execution(JScrollPane-Component Factory.create(String s))
12     { return new JScrollPane(proceed(s)); }
13   around2: call(void ExtPaintImpl-PaintImpl.paint())
14     { new ExtPaintImpl(target).proceed(); }
15   around3: execution(void PaintImpl.paint()) { proceed(); }
16 }
```

Fig. 3. Aspect for GUI component creation and display

Flexible type-safe advice with type ranges. Another key property of our approach is the flexible but safe mechanisms for the use of different argument and result types in advice and `proceed`. Such type variance is illustrated in Fig. 3 in the context of the creation and the display of a number of GUI components (lines 1–6). Here, GUI components may be created by the method `Factory.create` (line 8). The `around1` advice in aspect `A` (lines 11–12) overrides the factory method in order to add scrollbars to the created component. Mainstream aspect languages do not safely support such replacement advice: in AspectJ, `proceed` will always have the same type signature as the enclosing `around` method, which prohibits any type variance between the advice and `proceed`. Borrowing from earlier work on StrongAspectJ [4], our approach uses type ranges to permit a more flexible typing strategy: in the example, the typing of the result as a range `JScrollPane-Component` admits all join points with a return type within the range. In fact, the types in pointcuts are always type ranges. When we specify a singular type in the syntax this implicitly represents a type range with both the upper and lower bound equal to the single given type.

If we assume that the constructor of `JScrollPane` takes a `Component` argument, then we may observe that the advice behavior is indeed compatible with join points in this range. In StrongAspectJ, this is verified at the level of the type system by assigning `proceed` the return type `Component` (the upper bound) and requiring that the advice itself return a value of type `JScrollPane` (the lower bound). While this works for the given example, reducing type information to

the bounds entails a loss of type information which prevents the advice behavior from returning the original return value obtained from `proceed` (which now has type `Component`) as the return value of the advice method (which now requires type `JScrollPane`). For such a case of non-replacing advice behavior, StrongAspectJ supports so-called generic advice where explicit type variables are used to represent the return type of a join point. However, this mechanism is separated from the ordinary case, which means that one advice method cannot combine both behaviors (for example, as different branches of an if-test).

The contribution of the A calculus in this respect is that we integrate both mechanisms in one typing scheme that is lightweight for the programmer to use. The first advice method of aspect `A` is type-checked in the A calculus as follows: behind the scenes, the return type of the join point is represented by a fresh type variable, say Z, with lower bound `JScrollPane` and upper bound `Component`. `proceed` is assigned the return type Z, and the advice method should also return a value of type Z. Now, the upper and lower bound of Z enable a value of type Z to be converted to type `Component`, and enable a value of type `JScrollPane` to be converted to type Z. As such, the replacement advice behavior of this advice method is accepted by the type system. However, since there is no loss of type information, it remains possible to return the original return value from `proceed` as a result of the advice method (both have type Z)[3].

Receiver substitution. Fig. 3 also illustrates the concept of receiver substitution through one of its typical applications: the selection of different, non-modifiable implementations, used via a common interface. In the example, GUI components may be displayed by means of a `paint` method, which delegates to the `paint` method of an associated `PaintImpl` instance (line 3). The advice method `around2` (lines 13–14) replaces the receiver of the invocation of `PaintImpl.paint` with a more refined implementation named `ExtPaintImpl`, wrapping the original. This is accomplished by invoking the `proceed` closure (a static closure) with a new receiver constructed from the original receiver, which is provided to the advice body under the predefined name `target`. (This advice method is type checked similarly to the advice method `around1`, since this is also a replacement advice, although for the receiver rather than the return value.)

The concept of receiver substitution uncovers an important difference in our treatment of advice methods for `call` and `execution` join points. The conceptual difference between these join points is that method calls occur before method dispatch, whereas method execution occurs afterwards. Call advice is therefore typically used to intercept all general invocations of a given method, whereas execution advice is normally used to advise specific method implementations[4]. Receiver substitution is inherently problematic for execution advice, because the

[3] The use of fresh type variables to maintain type information is comparable to the process of *capture conversion* [18] used with wildcard types.

[4] In concrete implementations of aspect languages based on code manipulation, there is also the difference that call advice is usually woven at the call sites, while execution advice is woven into the called method itself. The corresponding difference in weaving overhead also impacts the choice between the two.

change of receiver may invalidate the criteria used to activate the advice in the first place. Indeed, the receiver determines which method implementation to call, and the method implementation determines whether a given advice is triggered.

There are a number of ways to treat receiver substitution for execution advice. AspectJ simply executes the chosen method implementation with the newly provided receiver. This is only type safe if the dynamic class of new receiver defines or inherits this method. If it can be enforced that the new receiver is an instance of a subclass of the class in which the chosen method implementation is defined, the type error is avoided. But we still believe this is a dangerous practice, since it circumvents dynamic dispatch and hence allows violation of class invariants of the subclass. For this reason, the A calculus only supports receiver substitution for call advice (where `proceed` is a static closure which takes a receiver) and prohibits it for execution advice (where `proceed` is a dynamic closure that only takes arguments, *e.g.*, `around3` on line 15).

```
1  aspect CFlow {
2    pointcut p(int dx):
3      execution(int Point.getX()) &&
4      cflow(execution(Point Point.move(int)) && args(dx));
5    int around(int dx): p(dx) {
6      System.out.println("getX in cflow of move("+dx+")");
7      return proceed(dx); }
8  }
9  class Point {
10   int x;
11   Point(int x) {}
12   int getX() { return x; }
13   Point move(int dx) { return new Point(getX()+dx); }
14 }
```

Fig. 4. Cflow example in AspectJ

Handling control-flow relationships. The selection of join points with a certain control-flow relationship, as expressed by the AspectJ pointcut designator `cflow`, is a common example of a non-trivial dynamic pointcut mechanism. This pointcut extension may be supported using the aforementioned advice selection parameters of our calculus, which could easily keep track of the call stack. However, for the particular case of `cflow`, we observe that it may alternatively be supported using the standard object-oriented features of our calculus, if the program is 'preprocessed' using an automatic transformation that represents the necessary call stack information as extra parameters of the method invocation.

The example in Figs. 4 and 5 illustrate this transformation. The first figure shows an application of an AspectJ `cflow` pointcut: executions of `Point.getX()` should only be intercepted if they occur in the control flow of the method

```
1  class Call { Call next; }
2  class Point_move extends Call { Point self; int dx; }
3  class CFlow {
4    Point_move select(Call c) {
5      if (c==null) return null;
6      else if (c instanceof Point_move) return (Point_move)c;
7      else return select(c.next);
8    }
9    around: execution(int Point.getX(Call c)) {
10     Point_move pm = select(c);
11     if (pm == null) return proceed();
12     System.out.println("getX in cflow of move("+pm.dx+")");
13     return proceed();
14   }
15 }
16 class Point {
17   int x;
18   int getX(Call c) { return x; }
19   Point move(Call c, int dx) {
20     return new Point(getX(new Point_move(this,dx))+dx);
21   }
22 }
```

Fig. 5. Cflow emulation

`Point.move(int)`. The corresponding advice just logs the corresponding call, and otherwise does not interfere with the base program's execution. The result of the transformation is shown in Fig. 5. The basic idea is that the call stack is represented by a list of objects provided as a newly added argument to every method call; the `cflow` semantics may then be implemented as a simple search through this list for an invocation of the requested method. Calls to the method `move` are represented by instances of the class `Point_move`, which extends the class `Call` that models calls in general. The `getX` method in class `Point` has been modified to include a `Call` parameter that represents the calling context, and the definition of the `move` method has been changed to provide this information as an instance of `Point_move` when it invokes `getX`. Finally, the aspect `Cflow` selects executions of the getter method as part of its pointcut, and uses the method `select` to test for a control-flow relationship on the `Call` parameter.

Pragmatic extensions. This section assumes several pragmatic extensions of the A calculus. These features are not present in the formal definition in Section 3, but they can be provided by fairly straightforward extensions. At the syntactic level, we made small changes such as leaving out the superclass when it is `Object`, and adding the keyword `return`. We also assumed the primitive types `int` and `void`, and we added a semicolon (to be used as $e;e'$ where e is evaluated and then ignored), and used `System.out.println(_)`. As mentioned, the

extension whereby advice selection matches on method names is easily covered by the advice selection parameters. The `cflow` code transformation involves `if` statements and `instanceof` expressions, but these extensions would not be hard to add. Finally, we have used anonymous closures depending on their lexical environment, whereas the A calculus only supports closures which bundle a method selector with either a class name or a receiver object. However, it is easy to transform an anonymous closure into such a pair. For instance, the following anonymous closure from Fig. 2:

```
int a, int b => proceed(a,s,b); return 0
```

can be expressed as ⟨`new B(proceed,s),m2`⟩ where `B` is a new class:

```
class B {
  (int,String,int)->int proceed;
  String s;
  int m2(int a, int b) { proceed(a,s,b); return 0; }
}
```

3 Formal Definition of the *A* Calculus

This section gives a formal definition of the A calculus. Following Featherweight Java by Igarashi *et al.* [16] the calculus omits many features, such as mutable state, concurrency, inner classes, reflection, interfaces, abstract methods, overloading, field shadowing, `super`, primitive types, access control, `null` references, and exceptions. Moreover, Featherweight Java includes casts for erasure analysis, but casts do not provide any significant insight here so we omit them.

Our notational conventions are generally well-known in the Featherweight Java context. Terminals are written in `teletype` and metavariables in *italic*. Metavariables represent syntactic categories; distinct metavariables may be used for the same syntactic category in rules or definitions in order to distinguish between multiple instances. Furthermore, we reserve the following metavariables for lexical categories: x for term variables (including `this`, `target`, `proceed`), X for type variables, m for method names, f, g for field names, C, D for class names (including `Object`), and a for advice names. The notation $\overline{e}$ represents the ordered sequence of elements $e_1, \ldots, e_n$ for some $n \geq 0$. Separators are implicit and will be determined by the context. Note that there is no overlap between $\overline{e}$ and e, so they may appear independently in the same definition. The sequence notation applies to binary constructs, too; for example, $\overline{P\ x}$ is a shorthand for $P_1\ x_1, \ldots, P_n\ x_n$, which also implies that both sequences have the same number of elements. Finally, we use the following standard constructors for sequence expressions: $\bullet$ is the empty sequence (*nil*), $e : \overline{e}$ is the sequence formed by the element e followed by $\overline{e}$ (*cons*), and $\overline{e_1}\ \overline{e_2}$ is the concatenation of $\overline{e_1}$ and $\overline{e_2}$.

3.1 Syntax

The syntax of the A calculus is defined in Fig. 6. Most syntax elements are available in user-level programs, but some are introduced for the purpose of type checking evaluation, as explained below.

Types can be either class names, variable types, static closure types, or dynamic closure types (the difference being that the latter do not include a receiver type). Declaration typing enforces that type variables cannot appear in programs, since there is no place where they can be declared. Instead, type variables are introduced in the type checking of advice methods.

Variables, object creation, and field access are standard, whereas method invocation is replaced by a combination of closure construction and closure application, for both static and dynamic closures. $\langle C, m\rangle$ denotes a static closure for method m on class C, while $\langle e, m\rangle$ denotes a dynamic closure for method m on receiver e. The corresponding closure applications are written $e\,.\,d(\overline{e})$ and $d(\overline{e})$, where d is the closure, e is the receiver (used with static closures only) and $\overline{e}$ are the arguments.

In order to model advice application, closure terms may be annotated with a list of advice. This list consists of pairs $e_i\,.\,a_i$, where a_i is the name of the advice method, and e_i is the corresponding aspect instance. Advice methods are class members and are therefore executed in the context of an aspect instance (accessible as `this` during the evaluation of the advice).

The syntax defines both term and type environments, represented by Γ and Δ respectively. A term environment maps term variables to types, while a type environment maps type variables to ranges S_i-U_i, where S_i is the lower bound for the type variable, and U_i the upper bound. While method declarations have a standard signature that consists of singular types for the result (T) and arguments ($\overline{T}$), advice methods are declared with type ranges for the result (S-U), intercepted receiver (S_0-U_0) and arguments ($\overline{S\text{-}U}$). These ranges contribute to

Type, term and value expressions

$$
\begin{array}{rcll}
S, T, U & ::= & C \mid X \mid S.\overline{T} \to T \mid \overline{T} \to T & \textit{types} \\
d, e & ::= & x \mid \texttt{new}\ C(\overline{e}) \mid e\,.\,f \mid e\,.\,d(\overline{e}) \mid d(\overline{e}) \mid \langle C,m\rangle \mid \langle C,m\rangle[\overline{e\,.\,a}] \mid & \textit{terms} \\
 & & \langle e,m\rangle \mid \langle e,m\rangle[\overline{e\,.\,a}] & \\
v & ::= & \texttt{new}\ C(\overline{v}) \mid \langle C,m\rangle \mid \langle C,m\rangle[\overline{v\,.\,a}] \mid \langle v,m\rangle \mid \langle v,m\rangle[\overline{v\,.\,a}] & \textit{values}
\end{array}
$$

Environments and auxiliaries

$$
\begin{array}{rcll}
\Gamma & ::= & \overline{T\ x} & \textit{var envs} \\
\Delta & ::= & \overline{S\text{-}U\ X} & \textit{tvar envs} \\
K & ::= & \texttt{call} \mid \texttt{exe} & \textit{adv. kinds}
\end{array}
$$

Member and top-level declarations

$$
\begin{array}{rcll}
M & ::= & T\ m(\overline{T\ x})\{e\} & \textit{methods} \\
A & ::= & a:\ K(S\text{-}U\ S_0\text{-}U_0\,.*(\overline{S\text{-}U\ x}))\{e\} & \textit{advice} \\
Q & ::= & \texttt{class}\ C\ \texttt{extends}\ D\ \{\overline{T\ f};\ \overline{M}\ \overline{A}\} & \textit{classes}
\end{array}
$$

Fig. 6. Syntax

determine the set of compatible join points, and they specify the bounds for type variables introduced during advice typing. The advice also declares a kind, which is either `call` or `exe`. The type ranges are placed in parentheses after the advice kind, in line with the pointcut syntax in AspectJ. Finally, class declarations specify a single parent class and a list of declarations of fields, ordinary methods, and advice methods. We omit constructors: they are included in Featherweight Java to make the language a subset of Java, but they contribute no information.

3.2 Dynamic Semantics

The evaluation rules of the A calculus are shown in Fig. 7. We first discuss some general issues, and then describe individual rules. The auxiliary function fields retrieves all fields, meth a method, and advice an advice with a certain name. The definitions are omitted, but standard. We assume the following standard sanity conditions: (i) there is at most one definition for each class name, (ii) there is no definition for `Object`, (iii) the parent of a class is either defined or `Object`, and (iv) inheritance is acyclic.

Evaluation uses judgments on the form $e, H \rightsquigarrow e', H'$, which is standard evaluation ($e \rightsquigarrow e'$) extended with an execution history mechanism. The history is updated by applying a history update function $[\![\_]\!]_{\_}$ to the current history, receiving the new term as an argument. E.g., the history H is transformed into $[\![H]\!]_{e'}$ by an evaluation step that produces the term e'. The history update function and the initial history value are free parameters of the calculus. Apart from the fact that the history update function must be total, they may be chosen freely in order to enable a particular type of advice selection. The congruence rule—omitted from Fig. 7 for brevity, but straightforward—updates the history just like the other rules. This means that the execution history at every step is updated using the complete program. It therefore has complete information about the evaluation and supports every conceivable advice selection mechanism.

Evaluation rules. Field access evaluation is standard (rule EvalField), but method invocation is covered by multiple rules, in order to handle closures and advice. Advice behavior is integrated into method invocation in three steps: (i) selection of applicable advice (ii) application of each advice method (iii) proceed to the next stage (or the original method invocation). This process must be run twice: once for call advice, where the receiver may still be updated, and once for execution advice, where the receiver is fixed. This results in the six remaining evaluation rules of Fig. 7. The different steps in the reduction from an application of a static closure to a method body may be illustrated as follows (omitting the execution history):

$$
\begin{array}{lcll}
e_0\,.\,\langle C, m\rangle(\overline{e}) & \rightsquigarrow & e_0\,.\,\langle C, m\rangle[\overline{e_a\,.\,a}](\overline{e}) & \textsc{EvalCAdvise} \\
 & \rightsquigarrow^* & e_0\,.\,\langle C, m\rangle[](\overline{e}) & \textsc{EvalCallInvk}^\dagger \\
 & \rightsquigarrow & \langle e_0, m\rangle(\overline{e}) & \textsc{EvalShift} \\
 & \rightsquigarrow & \langle e_0, m\rangle[\overline{e'_a\,.\,a'}](\overline{e}) & \textsc{EvalEAdvise} \\
 & \rightsquigarrow^* & \langle e_0, m\rangle[](\overline{e}) & \textsc{EvalExecInvk}^\dagger \\
 & \rightsquigarrow & e_b\,[\mathtt{this} \mapsto e_0, \overline{x} \mapsto \overline{e}] & \textsc{EvalInvk}
\end{array}
$$

$$\textsc{EvalField}\ \frac{\mathsf{fields}(C) = \overline{T\ f}}{\texttt{new}\ C(\overline{e})\,.\,f_i, H \rightsquigarrow e_i, [\![H]\!]_{e_i}}$$

$$\textsc{EvalInvk}\ \frac{e = \texttt{new}\ C(\cdot\cdot) \qquad \mathsf{meth}(C,m) = (\overline{x})\{e_0\} \qquad e' = e_0\,[\texttt{this} \mapsto e, \overline{x} \mapsto \overline{e}]}{\langle e, m\rangle[]\,(\overline{e}), H \rightsquigarrow e', [\![H]\!]_{e'}}$$

$$\textsc{EvalCAdvise}\ \frac{\mathsf{getCAdvice}(H, e_0, C, m, \overline{e}) = \overline{e_a\,.\,a} \qquad e' = e_0\,.\,\langle C, m\rangle[\overline{e_a\,.\,a}]\,(\overline{e})}{e_0\,.\,\langle C, m\rangle\,(\overline{e}), H \rightsquigarrow e', [\![H]\!]_{e'}}$$

$$\textsc{EvalEAdvise}\ \frac{e_0 = \texttt{new}\ C(\cdot\cdot) \qquad \mathsf{getEAdvice}(H, e_0, C, m, \overline{e}) = \overline{e_a\,.\,a} \qquad e' = \langle e_0, m\rangle[\overline{e_a\,.\,a}]\,(\overline{e})}{\langle e_0, m\rangle\,(\overline{e}), H \rightsquigarrow e', [\![H]\!]_{e'}}$$

$$\textsc{EvalCallInvk}\ \frac{e_a = \texttt{new}\ C_a(\cdot\cdot) \qquad \mathsf{advice}(C_a, a) = (\overline{x})\{e\} \qquad e' = e\,[\texttt{this} \mapsto e_a, \texttt{target} \mapsto e_0, \texttt{proceed} \mapsto \langle C, m\rangle[\overline{e_a\,.\,a}], \overline{x} \mapsto \overline{e}]}{e_0\,.\,\langle C, m\rangle[e_a\,.\,a :: \overline{e_a\,.\,a}]\,(\overline{e}), H \rightsquigarrow e', [\![H]\!]_{e'}}$$

$$\textsc{EvalShift}\ \frac{}{e_0\,.\,\langle C, m\rangle[]\,(\overline{e}), H \rightsquigarrow \langle e_0, m\rangle\,(\overline{e}), [\![H]\!]_{\langle e_0, m\rangle\,(\overline{e})}}$$

$$\textsc{EvalExecInvk}\ \frac{e_a = \texttt{new}\ C_a(\cdot\cdot) \qquad \mathsf{advice}(C_a, a) = (\overline{x})\{e\} \qquad e' = e\,[\texttt{this} \mapsto e_a, \texttt{target} \mapsto e_0, \texttt{proceed} \mapsto \langle e_0, m\rangle[\overline{e_a\,.\,a}], \overline{x} \mapsto \overline{e}]}{\langle e_0, m\rangle[e_a\,.\,a :: \overline{e_a\,.\,a}]\,(\overline{e}), H \rightsquigarrow e', [\![H]\!]_{e'}}$$

Fig. 7. Evaluation

The rule EvalCAdvise annotates the static closure with a list of call advice, while the rule EvalCallInvk is used zero or more times to remove and apply each advice. When the advice list is exhausted, rule EvalShift converts the static closure to a dynamic closure, where the process is repeated for execution advice, using rule EvalEAdvise and rule EvalExecInvk. When the advice list is again exhausted, rule EvalInvk reduces the term to the body of the given method, where the parameters are appropriately substituted. The dagger symbol (†) on EvalCallInvk and EvalExecInvk indicates that this example shows a particularly simple case: in general, these rules replace the application of an advised closure by an advice body of the first advice, where `proceed` is bound to the advised static closure, with the first advice removed. The advice body may or may not apply the `proceed` closure, and it may employ the original arguments or different ones. The above illustration shows the simple case where the advice body at some point directly applies `proceed` to the original arguments.

Advice selection. The aspect table, *AT*, is a globally available list of objects used as aspect instances, *i.e.*, they are candidates for matching advice whenever a closure invocation needs to be advised. We impose minimal restrictions on the aspect instances and require that only the outer expression is evaluated (so the instances are expressions of the form $\texttt{new}\ C(\overline{e})$). The rules EvalCAdvise and EvalEAdvise are used to select the list of applicable advice pairs using the

functions getCAdvice and getEAdvice. These functions are free parameters of the calculus, i.e., they may be chosen as needed in order to express the desired advice selection semantics. However, for soundness, they must satisfy the following requirement. We include the previously mentioned requirement on the history update function, in order to have a complete list of requirements in one place:

Requirement 1 (The A calculus parameters). *The history update function* $[\![\_]\!]\_$ *and the functions* getCAdvice *and* getEAdvice *must be total. Moreover, whenever* $\mathsf{getCAdvice}(H, e_0, C, m, \overline{e}) = \overline{e_a\,.\,a}$ *or* $\mathsf{getEAdvice}(H, e_0, C, m, \overline{e}) = \overline{e_a\,.\,a}$*, it must be the case that* $\vdash \overline{e_a\,.\,a}$ appropriate for $K\ C\,.\,m$*, where respectively* $K = \texttt{call}$ *and* $K = \texttt{exe}$*.*

Fig. 8 defines what it means for one advice pair to be "appropriate for" a particular advice kind K and join point $C\,.\,m$: the rule AppAdvice specifies that the aspect class must contain an advice method of the appropriate kind, and with a type compatible with the type of the given join point $C\,.\,m$. Advice type compatibility ($\sqsubset$) is further defined in Fig. 9.

$$\textsc{AppAdvice}\quad \frac{\begin{array}{l}\texttt{new}\ C_a(\overline{e}) \in AT \qquad \mathsf{advice}(C_a, a) = K(S\text{-}U\ S_0\text{-}U_0\,.\,*(\overline{S\text{-}U}))\{\cdot\cdot\} \\ \mathsf{meth}(C, m) = T\ (\overline{T\ x})\{\cdot\cdot\} \qquad C.\overline{T} \to T \sqsubset S_0\text{-}U_0.\overline{S\text{-}U} \to S\text{-}U\end{array}}{\vdash \texttt{new}\ C_a(\overline{e})\,.\,a\ \mathsf{appropriate\ for}\ K\ C\,.\,m}$$

Fig. 8. Appropriate advice pairs

Note that this open declaration of advice selection allows the functions getCAdvice and getEAdvice to use the execution history (H), and it allows them to produce a list of advice of the appropriate kind which may be ordered independently of the order of aspects in AT, and may even contain duplicates. This enables modeling of dynamic advice or aspect activation and deactivation, as well as arbitrary advice precedence schemes. Essentially, appropriate advice is all that is required for safety.

History counting example. To illustrate the instantiation of calculus parameters, we consider a very simple history-based pointcut that selects a certain call advice method a on advice instance e_a for every nth evaluation step. The execution history is modeled as an integer number (i), and the history update and advice selection functions are defined as follows (where div is a divisibility predicate):

$$[\![i]\!]\_ = i + 1 \qquad \begin{array}{l}\mathsf{getCAdvice}(i, \_, \_, \_, \_) = \text{if } i \ \mathsf{div}\ n \text{ then } e_a\,.\,a \text{ else } \bullet \\ \mathsf{getEAdvice}(\_, \_, \_, \_, \_) = \bullet\end{array}$$

In case the type of advice method a is compatible with any method type in the class table, this instantiation satisfies Requirement 1.

3.3 Static Semantics

The static semantics of the A calculus consists principally of a definition of subtyping, term typing and declaration typing.

$$\textsc{SubRefl}\ \ \Delta \vdash T <: T$$

$$\textsc{SubTrans}\ \ \frac{\Delta \vdash T <: T'' \qquad \Delta \vdash T'' <: T'}{\Delta \vdash T <: T'}$$

$$\textsc{SubExtends}\ \ \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{\cdot\cdot\}}{\Delta \vdash C <: D}$$

$$\textsc{SubScl}\ \ \frac{\Delta \vdash S' <: S \qquad \Delta \vdash \overline{T' <: T} \qquad \Delta \vdash T <: T'}{\Delta \vdash S.\overline{T} \to T <: S'.\overline{T'} \to T'}$$

$$\textsc{SubDcl}\ \ \frac{\Delta \vdash \overline{T' <: T} \qquad \Delta \vdash T <: T'}{\Delta \vdash \overline{T} \to T <: \overline{T'} \to T'}$$

$$\textsc{SubUp}\ \ \frac{\Delta(X) = S\text{-}U}{\Delta \vdash X <: U}$$

$$\textsc{SubLow}\ \ \frac{\Delta(X) = S\text{-}U}{\Delta \vdash S <: X}$$

$$\textsc{CompatAdvice}\ \ \frac{\vdash S <: T \qquad \vdash T <: U \qquad \vdash \overline{S <: T} \qquad \vdash \overline{T <: U} \qquad \vdash S_0 <: T_0 \qquad \vdash T_0 <: U_0}{T_0.\overline{T} \to T \sqsubset S_0\text{-}U_0.\overline{S\text{-}U} \to S\text{-}U}$$

$$\textsc{OkObj}\ \ \Delta \vdash \texttt{Object}$$

$$\textsc{OkClass}\ \ \frac{\texttt{class}\ C\ \{\cdot\cdot\}}{\Delta \vdash C}$$

$$\textsc{OkVar}\ \ \frac{X \in \mathrm{dom}(\Delta)}{\Delta \vdash X}$$

$$\textsc{OkScl}\ \ \frac{\Delta \vdash S, \overline{T}, T}{\Delta \vdash S.\overline{T} \to T}$$

$$\textsc{OkDcl}\ \ \frac{\Delta \vdash \overline{T}, T}{\Delta \vdash \overline{T} \to T}$$

Fig. 9. Subtyping, advice compatibility and well-formed types

Preliminaries. Fig. 9 presents subtyping, advice compatibility, and type well-formedness. Subtyping is reflexive and transitive, and includes subclassing and standard function type rules for closures (rule SubRefl through rule SubDcl). Type variables are related to their bounds (SubUp and SubLow). Advice compatibility between a singular type signature and a range type signature is defined by range inclusion (CompatAdvice). Finally, type well-formedness (OkObj through OkDcl) just requires that all class names and type variables are defined.

Term typing. The term typing rules, one for each syntactic shape, are presented in Fig. 10. Some rules use bound_Δ, which replaces a type variable by its upper bound in Δ, and otherwise retains the type unmodified. For typing rules that do not have any typing judgments as premises, we require a well-formed term environment, $\Gamma\ \mathsf{ok}$, i.e., that it does not contain multiple bindings for the same variable. As such, term typing will imply a well-formed term environment.

Note that the typing of advised closures, rules TSclA and TDclA, require a typing of the advice. Advice typing is defined in rule ATChk: it requires that the advice method is of the correct advice kind (K) and that its signature is compatible with that of the join point. Advice typing is needed for soundness, to ensure that advice application preserves the type of the advised invocation.

Declaration typing. The typing rules for member declarations in class C with parent class D are specified in Fig. 11. For method declarations, rule OkMethod requires that types are well-formed, the advice body is typable, and overriding is invariant. For advice declarations, rule OkAdvice uses a type environment containing fresh type variables with the declared ranges, and a term environment

$$\textsc{TVar}\ \frac{\Gamma\ \mathsf{ok} \qquad \Gamma(x) = T}{\Delta;\Gamma \vdash x : T} \qquad \textsc{TField}\ \frac{\Delta;\Gamma \vdash e : T_0 \qquad \mathsf{bound}_\Delta(T_0) = C \qquad \mathsf{fields}(C) = \overline{T\ f}}{\Delta;\Gamma \vdash e.\ f_i : T_i} \qquad \textsc{TNew}\ \frac{\Gamma\ \mathsf{ok} \qquad \mathsf{fields}(C) = \overline{T\ f} \qquad \Delta;\Gamma \vdash \overline{e : U} \qquad \Delta \vdash \overline{U <: T}}{\Delta;\Gamma \vdash \mathtt{new}\ C(\overline{e}) : C}$$

$$\textsc{TInvk}\ \frac{\Delta;\Gamma \vdash d : T_0 \qquad \mathsf{bound}_\Delta(T_0) = S.\overline{T} \to T \qquad \Delta;\Gamma \vdash e_0 : S' \qquad \Delta \vdash S' <: S \qquad \Delta;\Gamma \vdash \overline{e : T'} \qquad \Delta \vdash \overline{T' <: T}}{\Delta;\Gamma \vdash e_0.\ d(\overline{e}) : T} \qquad \textsc{TCInv}\ \frac{\Delta;\Gamma \vdash d : T_0 \qquad \mathsf{bound}_\Delta(T_0) = \overline{T} \to T \qquad \Delta;\Gamma \vdash \overline{e : T'} \qquad \Delta \vdash \overline{T' <: T}}{\Delta;\Gamma \vdash d(\overline{e}) : T}$$

$$\textsc{TScl}\ \frac{\Gamma\ \mathsf{ok} \qquad \mathsf{meth}(C, m) = T\ (\overline{T\ x})\{\cdot\cdot\}}{\Delta;\Gamma \vdash \langle C, m\rangle : C.\overline{T} \to T} \qquad \textsc{TDCl}\ \frac{\Delta;\Gamma \vdash e_0 : S \qquad \Delta \vdash S <: C \qquad \mathsf{meth}(C, m) = T\ (\overline{T\ x})\{\cdot\cdot\}}{\Delta;\Gamma \vdash \langle e_0, m\rangle : \overline{T} \to T}$$

$$\textsc{TSclA}\ \frac{\mathsf{meth}(C, m) = T\ (\overline{T\ x})\{\cdot\cdot\} \qquad \Delta;\Gamma \vdash \overline{e.a} : \mathtt{call}\ C.\overline{T} \to T}{\Delta;\Gamma \vdash \langle C, m\rangle[\overline{e.a}] : C.\overline{T} \to T} \qquad \textsc{TDClA}\ \frac{\Delta;\Gamma \vdash e_0 : S \qquad \Delta \vdash S <: C \qquad \mathsf{meth}(C, m) = T\ (\overline{T\ x})\{\cdot\cdot\} \qquad \Delta;\Gamma \vdash \overline{e.a} : \mathtt{exe}\ C.\overline{T} \to T}{\Delta;\Gamma \vdash \langle e_0, m\rangle[\overline{e.a}] : \overline{T} \to T}$$

$$\textsc{ATChk}\ \frac{\Delta;\Gamma \vdash e : T_a \qquad \Delta \vdash T_a <: C_a \qquad \mathsf{advice}(C_a, a) = K(S\text{-}U\ S_0\text{-}U_0.*(\overline{S\text{-}U}))\{\cdot\cdot\} \qquad T_0.\overline{T} \to T \sqsubset S_0\text{-}U_0.\overline{S\text{-}U} \to S\text{-}U}{\Delta;\Gamma \vdash e.a : K\ T_0.\overline{T} \to T}$$

Fig. 10. Term typing, with auxiliary advice typing

$$\textsc{OkMethod}\ \frac{\vdash T, \overline{T} \qquad \bullet; C\ \mathtt{this}, \overline{T\ x} \vdash e : S \qquad \vdash S <: T \qquad \text{if } \mathsf{meth}(D, m) = T'\ (\overline{T'\ x'})\{\cdot\cdot\} \text{ then } T' = T \text{ and } \overline{T' = T}}{T\ m(\overline{T\ x})\{e\}\ \mathsf{ok\ for}\ C, D}$$

$$\textsc{OkAdvice}\ \frac{Y, X_0, \overline{X} \text{ distinct} \qquad \Delta = S\text{-}U\ Y, S_0\text{-}U_0\ X_0, \overline{S\text{-}U\ X} \qquad \Delta; C\ \mathtt{this}, X_0\ \mathtt{target}, \mathsf{pcd}(K, X_0, \overline{X}, X)\ \mathtt{proceed}, \overline{X\ x} \vdash e : T' \qquad \Delta \vdash T' <: Y \qquad \vdash S, U, S_0, U_0, \overline{S, U} \qquad \mathsf{advice}(D, a) \text{ undefined}}{a :\ K(S\text{-}U\ S_0\text{-}U_0.*(\overline{S\text{-}U\ x}))\{e\}\ \mathsf{ok\ for}\ C, D}$$

$$\mathsf{pcd}(\mathtt{call}, S, \overline{T}, T) = S.\overline{T} \to T \qquad \mathsf{pcd}(\mathtt{exe}, S, \overline{T}, T) = \overline{T} \to T$$

Fig. 11. Class member typing

including `this`, `target`, and `proceed` in addition to the arguments. The type of `proceed` is a static closure type for a call advice and a dynamic closure type for an execution advice, as determined by `pcd`, which is also defined in Fig. 11. Finally, a well-formed class declaration requires well-formed types, methods, and advice methods, and distinct declared names. The rule is straightforward and omitted for brevity.

3.4 Evaluation of Design Decisions

We now revisit some of the highlighted contributions from Section 1 and the discussion from Section 2, and we connect this to the formal presentation of the calculus from this section.

As explained in Section 2, the *A* calculus shifts some concepts typically considered aspect-oriented to the level of plain object-oriented programming. Most notably, `proceed` is a simple term variable that denotes a closure (see EvalCallInvk and EvalExecInvk in Fig. 7). Closures, static as well as dynamic, are first-class entities in the language (see Fig. 6), and fully integrated in the type system (see, e.g., rules TScl and TDcl in Fig. 10). Programs can create closures as shown in advice `around2` of Fig. 2, pass them to ordinary object-oriented methods (`m` in the example), and thus use well-known object-oriented reuse and abstraction mechanisms in the implementations of advice behavior.

Furthermore, we argue in Section 2 that advice for call and execution join points should be treated differently, and that receiver substitution should only be allowed for call advice, which should occur before execution advice. This distinction is enforced in the type checking from Fig. 11, where the function `pcd` returns a static closure type for advice declarations of kind `call`, and a dynamic closure type for ones of kind `exe`. In the execution of Fig. 7, appropriate static and dynamic closures are also used, and since rule EvalShift is the only rule that allows the switch from one to the other, the relative ordering between call and execution advice is guaranteed.

Next, the *A* calculus has an advice selection parameter that takes the place of any concrete advice selection strategy. This is represented in the formal specification of the calculus by the functions getCAdvice and getEAdvice of Fig. 7 for which no definition is given, and which are only subject to the stipulations of Requirement 1 and Fig. 8. These functions obviously include selection of join points based on method patterns (as used in Section 2), as well as dynamic activation and advice precedence. In addition, advice selection may be based on the execution history, which is represented by H and manipulated by the history update function $[\![\_]\!]\_$, again parameters for which the definition is left open.

Finally, the *A* calculus ensures type safety using the simplified notion of type ranges, which allows for simpler and more intuitive advice declarations. Whereas De Fraine *et al.* [4] propose two mechanisms (ordinary advice and generic advice) to support two classes of advice behavior, they are now integrated in one mechanism where type variables are only used in the type checking process. While the syntax of Fig. 6 includes type variables, they are not allowed to appear in the syntax of programs since declaration typings such as rule OkMethod in Fig. 11

require a typing using an empty type environment; since term typing requires well-formed types, this excludes type variables because they are not well-formed in an empty type environment (see rule OkVar in Fig. 9). Contrarily, the type checking of advice declarations with rule OkAdvice in Fig. 11 employs type variables $Y, X_0, \overline{X}$, but these are not specified as a part of the advice declaration. The type variables have both an upper bound and lower bound in the type environment that is used in this rule. The implied subtype relations (through both rules SubUp and SubLow from Fig. 9) are crucial to enable replacement advice such as method `around1` from Fig. 3. For non-replacement advice, the types do not need to be converted since both the types of `proceed` on the one hand, and those of the other arguments and the result of the advice on the other hand, are directly expressed using one of the type variables $Y, X_0, \overline{X}$.

4 Safety Properties

We now present the safety properties of the A calculus. Under the assumption that all defined classes are well-formed, and the aspect instances in the aspect table (AT) are well-typed, the following preservation and progress properties hold:

Theorem 2 (Preservation). *If* $\bullet; \Gamma \vdash e : T$ *and* $e, H \rightsquigarrow e', H'$*, then* $\bullet; \Gamma \vdash e' : T'$ *for some* T' *such that* $\vdash T' <: T$

Theorem 3 (Progress). *If* $\bullet; \bullet \vdash e : T$ *then either* e *is a value or* $e, H \rightsquigarrow e', H'$ *for any* H *and some* e' *and* H'*.*

The progress result employs the notion of a *value* as defined in Fig. 6.

We have developed a proof of these properties using the Coq proof assistant [19]. The proof is available at the web address given at the end of Section 1. Since the proof uses a classical structure, and since it has been mechanically verified, we will not present it in detail here. The calculus parameters, used to abstract over a concrete advice selection strategy, are represented as unknown sets and functions in the Coq specification, along with a statement that Requirement 1 is assumed. For preservation, we can obtain the necessary relations directly from Requirement 1, and from the rule AppAdvice in Fig. 8. For progress, the functions getCAdvice and getEAdvice must be total, which is obtained from Requirement 1, and similarly for the history update and advice selection functions.

Two design decisions have simplified the proof compared to previous approaches; we therefore briefly discuss these decisions and their impact. First, it is well-known that the modeling of *capture-avoiding substitution* in formal metatheory is a rather complex issue (this problem, and different solutions, were recently discussed by Aydemir *et al.* [20]). In our development, we can avoid these issues due to the fact that binders in the language cannot be nested, which removes the danger of variable capture during substitution. In particular, we do not include anonymous closures in the language. (For a discussion of how to emulate anonymous closures, see the end of Section 2).

Secondly, the use of type variables with both lower and upper bounds poses certain problems with respect to typical properties. For example, if the type

environment contains unsatisfiable assumptions such as $\Delta = \texttt{Number-String}\ X$, then it is possible to derive subtype relations that do not correspond to the class hierarchy, for example $\Delta \vdash \texttt{Number} <: \texttt{String}$. Rather than introducing well-formedness criteria for type environments that preclude these situations, we observe that it is possible to exploit that these constraints are unsatisfiable. For example, we may define the following advice method, which, because of the type range `Number-String`, may invoke a `String` method on a `Number` instance in its body:

```
around: call(void Number-String.*()) { 12.length(); }
```

However, this is not a problem for the type safety of an actual program in the A calculus, since we know that this advice body is 'dead code': the advice cannot be applied because—barring any further assumptions—there exists no receiver of a type compatible with the range `Number-String`.

Note that, as a consequence, the preservation property is formulated using an empty type environment, which differs from the literature: Featherweight Generic Java [16], for instance, preserves term typing in both non-empty type and non-empty term environments (of course, it only supports upper bounds for type variables so these issues do not arise). In the A calculus, such a general preservation result does not hold, for the aforementioned reasons. However, it is crucial that the specialized preservation property that we prove is sufficient for type safety: program evaluation starts with a main expression that is well-typed in an empty type environment, and as the reduction of this main expression proceeds, method and advice bodies are only introduced after substituting expressions for their variables (where the standard *substitutivity* lemma ensures well-typedness), and the type environment remains empty. Note that if we had had type variable binders in expressions, they would introduce non-empty environments and thereby complicate proofs by induction. Given that this is not so common, we find it likely that the use of empty type environments for preservation in other calculi may also simplify their proofs.

5 Related Work

Our work is related to a number of different efforts within the AOSD and related communities. In this section we compare our work with respect to three groups of related work that share part of our main motivation, providing a simple and regular, yet expressive foundational calculus. Concretely, we consider three classes of related work:

1. Formal semantics and calculi for AspectJ-like aspect models.
2. Aspect models that are more general than the mainstream (AspectJ) model.
3. Approaches striving for the integration of aspects and objects.

There are several approaches using closures to model advice and `proceed`, *e.g.*, because it is an obvious choice in the context of a functional language, but we do not discuss these in detail here because they have different goals and the comparison does not bring significant insight.

Formal semantics and calculi for AspectJ-like aspect models. Work on formal AspectJ-like aspect models has started with the denotational semantics of AspectJ proposed by Wand *et al.* [13] which provides a detailed account of the pointcut and advice mechanisms of AspectJ. Furthermore, the authors have pointed out the basic type soundness issues of pointcut/advice binding (without providing a corresponding sound type system).

MiniMAO_1 [14] provides a very faithful model of the semantics of AspectJ, even including receiver substitution in advice. In contrast to our work, no rationale and formal justification is provided as to how call and execution pointcuts have to be handled in the presence of advice chains. Finally, MiniMAO_1 only provides a very restrictive type system that almost completely forbids type variability in the presence of `proceed`. We provide much more flexibility, most notably by supporting generic advice behavior.

Jagadeesan *et al.* [15] investigate the integration of pointcut/advice bindings with generics, providing sound type systems for aspect programs in the presence of a type-carrying and type-erasure semantics. They provide, however, less general pointcut and advice models. There is no receiver substitution by advice, and their pointcut language is limited (it does not include negation, for example). In contrast to our approach, their type system supports only limited type variance for `proceed` and relies on explicit type variables for the handling of generic advice.

StrongAspectJ [4] provides a more general model for type safe pointcut/advice bindings that supports type variability in the presence of `proceed`. In contrast to the previous systems, this approach also permits the correct handling of type variance for generic as well as non-generic advice: it enables, in particular, replacement advice to be typed correctly. Furthermore, it accommodates more general pointcut languages by means of a simple criterion of compatibility between pointcuts and advice. However, our parametrized type system supports a more general set of pointcut languages. Also, StrongAspectJ requires explicit type variables to enable generic advice behavior: our approach requires fewer type variables to be specified explicitly. Finally, StrongAspectJ does not specifically consider receiver substitution and call/execution join points, and handles them identical to AspectJ.

In addition to the limitations discussed above, all of these approaches do not strive, as we do, for a simpler model that attempts to model the ad hoc mechanisms of AspectJ in a regular fashion. In particular, none of these approaches shift complexity from the aspect model to well-understood object-oriented abstractions. Furthermore, since all of these approaches are centered on basic AspectJ features, none of them considers control-flow relationships, aspect enabling and advice redefinition. Finally, no mechanically checked type soundness proofs have been published for these earlier systems (though such a proof has recently been developed for StrongAspectJ [21].)

Beyond the AspectJ model. A significant number of papers have proposed generalized and extended versions of the pointcut languages and more powerful aspect models than that of AspectJ.

State-based aspects [5,6] feature aspects that are defined using finite-state based relationships over the history of program executions that are formalized in terms of an operational semantics. Similarly, tracematches [7] define pointcuts using a form of regular expressions over execution traces. Other approaches [8,9,10,11] have considered even more expressive pointcut languages, mostly defined in terms of functional or logical expressions, and some even Turing-complete. While we do not provide explicit representations for such pointcut languages, our parameter-based handling of advice selection allows them to be defined based on our model.

Furthermore, Douence *et al.* [6] exploit state-based aspects for the definition of a flexible notion of aspect activation in terms of regular activation conditions. This notion can also be modeled based on our parameter-based approach to advice selection.

Several formal and typed models for AOP have been developed that present advice application models different from the AspectJ model. Lämmel [12] presents a big-step operational semantics for the interception of method calls at so-called dispatch, enter and exit join points. In his most advanced system, advice, i.e. superimposed methods, may, similar to pointcuts, abstract over multiple intercepted methods. The corresponding language is shown to be type safe (without a mechanically checked proof). His language does not support the integration of advice execution with plain method calls as we permit using first-class closures. Furthermore, type variance is limited because no lower bounds may be taken into account as part of advice execution. In a functional setting, Ligatti *et al.* [22] provide a calculus with first-class advice that is triggered and may return to explicitly labeled program locations. Aldrich [23] has introduced a model of how advice application may interact with a strongly encapsulating module system. Both approaches, however, do not include basic object-oriented features, such as object subtyping and type variance.

In contrast to our work, these approaches do not aim at a general aspect theory. Moreover, while several strive for improved expressiveness and regularity in the definition of pointcuts and certain (limited) characteristics of advice execution, all systems, with the notable exception of Lämmel's, exclusively do so in functional or logical settings: all are more limited than our approach in shifting complexity from aspects to objects as we propose.

Integration of aspects and objects. Only few works have been devoted to the better integration between aspect-oriented and object-oriented concepts and mechanisms, in particular in order to investigate the use of abstraction mechanisms from both domains in order to structure aspect programs.

Ernst and Lorenz [24] have explored a notion of aspectual polymorphism and its interaction with plain object-oriented features. They have shown, in particular, that late binding of advice together with a notion of advice grouping can be used to introduce late binding even in base languages with early binding semantics. This work is therefore mainly concerned with a challenge that is complementary to ours: the expressiveness of the advice mechanism is increased in order to enable abstraction mechanisms known from object-oriented

programming to be applied on the aspect level. In contrast, we strive for the use of OO abstraction mechanism to simplify aspect abstractions.

Aspectual collaborations [25] (an extension of adaptive plug-and-play components [26]) consist of classes that may contain aspectual methods. Groups of ordinary methods that are intercepted by an aspectual method are identified by means of explicit constraints on the formal return and argument types of that aspectual method. An attachment mechanism between aspectual methods and ordinary ones instantiates the formal types with respect to the intercepted method, thus supporting type safety and variance in the Java sense. Compared to our approach, integration of aspects and objects is clumsy because calls to intercepted method have to be performed through a reflection interface. Furthermore, type variance is limited because lower bounds cannot be defined and the compatibility relation between aspectual methods and intercepted methods is stricter than in our approach.

6 Conclusions

In this paper we have presented the A calculus, a new foundation for aspect-oriented programming, that is simpler, more regular, but also more expressive than previous approaches. Simplicity and regularity is mainly supported by a better integration of aspect-oriented and object-oriented concepts, which has enabled us to shift complexity from aspects to well-known OO abstractions. We have, in particular, modeled the `proceed` mechanism using first-class closures and exploiting standard object-oriented abstractions to structure pointcuts and advice. The A calculus provides more expressiveness (i) by making explicit the significant differences on the semantic level of call and execution pointcuts especially in the presence of receiver substitution, and (ii) by supporting a large design space for non-essential features, such as advanced pointcut mechanisms and notions of aspect enabling and redefinition. Moreover, we have shown the complete calculus as well as all design choices for non-essential features to be type safe. Despite its expressiveness, the type system itself is more user-friendly than previous systems in that type ranges have to be specified in limited contexts and type variables are completely hidden from users. Finally, the type soundness of our system has been checked mechanically in Coq.

This work paves the way for different tracks of future work. Most directly, the design spaces for aspect features such as advanced pointcut mechanisms, aspect activation and aspect inheritance should be explored. More fundamental and longer term benefits of the calculus can be expected to consist in the definition of easier-to-use and semantically cleaner aspect languages that interact more smoothly with object-oriented base languages.

References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)

2. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
3. Brichau, J., et al.: Survey of aspect-oriented languages and execution models. Deliverable 12, Project IST-2-004349-NOE "AOSD-Europe" (May 2005)
4. De Fraine, B., Südholt, M., Jonckers, V.: StrongAspectJ: Flexible and safe pointcut/advice bindings. In: Mezini, M. (ed.) Proceedings AOSD 2008, pp. 60–71. ACM Press, New York (March 2008)
5. Douence, R., Fradet, P., Südholt, M.: A framework for the detection and resolution of aspect interactions. In: Batory, D., Consel, C., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 173–188. Springer, Heidelberg (2002)
6. Douence, R., Fradet, P., Südholt, M.: Composition, reuse and interaction analysis of stateful aspects. In: Lieberherr, K. (ed.) Proceedings AOSD 2004, pp. 141–150. ACM Press, New York (March 2004)
7. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. In: Johnson, R.E., Gabriel, R.P. (eds.) Proceedings OOPSLA 2005, pp. 345–364. ACM Press, New York (2005)
8. Douence, R., Motelet, O., Südholt, M.: A formal definition of crosscuts. In: Yonezawa, A., Matsuoka, S. (eds.) REFLECTION 2001. LNCS, vol. 2192, pp. 170–186. Springer, Heidelberg (2001)
9. Gybels, K., Brichau, J.: Arranging language features for pattern-based crosscuts. [27] 60–69
10. Walker, R.J., Viggers, K.: Implementing protocols via declarative event patterns. In: Proc. ACM SIGSOFT Int' Symp. on Foundations of Software Engineering (FSE-12), pp. 159–169. ACM Press, New York (2004), also TR no. 2004-745-10, Uni. of Calgary, `http://lsmr.cpsc.ucalgary.ca/papers/walker-fse-2004.pdf`
11. Ostermann, K., Mezini, M., Bockisch, C.: Expressive pointcuts for increased modularity. [28] 214–240
12. Lämmel, R.: A semantical approach to method-call interception. In: Kiczales, G. (ed.) Proceedings AOSD 2002, pp. 41–55. ACM Press, New York (April 2002)
13. Wand, M., Kiczales, G., Dutchyn, C.: A semantics for advice and dynamic join points in aspect-oriented programming. Trans. on Programming Languages and Systems (TOPLAS) 26(5), 890–910 (2004)
14. Clifton, C., Leavens, G.T.: MiniMAO1: An imperative core language for studying aspect-oriented reasoning. Science of Computer Programming 63(3), 321–374 (2006)
15. Jagadeesan, R., Jeffrey, A., Riely, J.: Typed parametric polymorphism for aspects. Science of Computer Programming 63(3), 267–296 (2006)
16. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. Transactions on Programming Languages and Systems (TOPLAS) 23(3), 396–450 (2001)
17. Odersky, M., Sulzmann, M., Wehr, M.: Type inference with constrained types. Theory and Practice of Object Systems 5(1), 35–55 (1999)
18. Torgersen, M., Hansen, C.P., Ernst, E., von der Ahé, P., Bracha, G., Gafter, N.M.: Adding wildcards to the Java programming language. In: Haddad, H., Omicini, A., Wainwright, R.L., Liebrock, L.M. (eds.) Proc. of the 2004 ACM Symposium on Applied Computing (SAC), pp. 1289–1296. ACM Press, New York (2004)
19. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science, vol. XXV. Springer, Heidelberg (2004)

20. Aydemir, B., Charguéraud, A., Pierce, B.C., Pollack, R., Weirich, S.: Engineering formal metatheory. In: Proceedings POPL 2008, pp. 3–15. ACM Press, New York (2008)
21. De Fraine, B.: Language Facilities for the Deployment of Reusable Aspects. PhD thesis, Vrije Universiteit Brussel (June 2009), http://soft.vub.ac.be/soft/_media/members/brunodefraine/phd.pdf
22. Ligatti, J., Walker, D., Zdancewic, S.: A type-theoretic interpretation of pointcuts and advice. Science of Computer Programming 63(3), 240–266 (2006)
23. Aldrich, J.: Open modules: Modular reasoning about advice [28] 144–168
24. Ernst, E., Lorenz, D.H.: Aspects and polymorphism in Aspect J. [27] 150–157
25. Lorenz, D.H., Lieberherr, K., Ovlinger, J.: Aspectual collaborations: Combining modules and aspects. The Computer Journal 46(5), 542–565 (2003)
26. Mezini, M., Lieberherr, K.: Adaptive plug-and-play components for evolutionary software development. In: Proceedings OOPSLA 1998, pp. 97–116. ACM Press, New York (October 1998)
27. Akşit, M. (ed.): Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003). ACM Press, New York (March 2003)
28. Black, A.P. (ed.): ECOOP 2005. LNCS, vol. 3586. Springer, Heidelberg (2005)