Fachbereich Informatik

Lehrgebiet Programmiersysteme

Prof. Dr. Friedrich Steimann

Abschlussarbeit im Studiengang Master of Computer Science

Implementation einer Erweiterung von Java um implizite Aufrufe mit impliziter Ankündigung

Implementing a Java extension supporting
implicit invocation with implicit announcement

Vorgelegt von:

Thomas Pawlitzki

Friedrichstr. 25

51143 Köln

Matrikelnummer: 7074573

Köln, den 02.07.2007

Abstrakt

Um die Qualität und die Effizienz in der Entwicklung von Software zu erhöhen, werden immer wieder neue Programmierparadigmen entworfen oder bestehende weiter entwickelt. Eine Weiterentwicklung der objektorientierten Programmierung ist die aspektorientierte Programmierung. Das Ziel der aspektorientierte Programmierung ist es, quer schneidende Funktionalität (wie z. B. Logging oder Sicherheit) zu modularisieren.

Eine aspekorientierte Erweiterung für die Programmiersprache Java ist AspectJ. Kritisiert wird an diesem Ansatz, dass es keinerlei Restriktionen gibt, wie AspectJ den Programmabfluss verändern und beeinflussen kann. Dies kann zu einer erschwerten Lesbarkeit und Nachvollziehbarkeit führen. Außerdem kann die Kopplung zwischen Aspekten und Klassen hoch sein, da Aspekte u. U. die Interna einer Klasse kennen müssen, um ihre Logik mit der der Klassen zu kombinieren.

Diese Arbeit beschäftigt sich mit der Integration einer auf AspectJ basierende Erweiterung für Java. Diese Erweiterung integriert implizite Aufrufe mit impliziter Ankündigung in die Programmiersprache. Diese Erweiterung versucht Aspekte und Klassen zu entkoppeln, indem eine Schnittstelle zwischen den beiden Komponenten eingeführt wird. Weiterhin dokumentiert diese Arbeit die Implementierung des Compilers, der Java um implizite Aufrufe mit impliziter Ankündigung erweitert.

Structure

List of figures

List of tables

Listings

Prologue

In order to improve quality and efficiency in software development the continuous development and research of new or the enhancement of existing programing paradigms is necessary. A development of the object oriented programming is the aspect oriented programming (abbreviated AOP). One of the main goals of AOP is the modularization of crosscutting concerns, which can be subsumed under the idea of separation of concerns.

In asymmetric approaches of AOP (to which AspectJ is counted) concerns are divided into two types: core concerns and crosscutting concerns. Core concerns realize the core functionality (business logic) of a software system, whereas crosscutting concerns involve system wide functionality as logging or security.[AJIA2003] For crosscutting concerns it is characteristic that they cannot be separated cleanly from other ones. So an implementation of an crosscutting concern by using object oriented languages causes code scattering and code tangling[1].

When modularizing crosscutting concerns with AspectJ it sometimes is criticized that aspects are able to influence and change almost unrestrictedly the normal control flow of a program. Furthermore an aspect is able to manipulate classes without the classes being aware of the changes. So the readability, traceability, debugging, and understanding of programs becomes more complicated. In order to change the behavior of classes the aspects need often internal knowledge of the classes. In addition to this, the fact that classes are not aware of the changes makes refactoring within the classes complicated, because the developer has to recognize, which aspects are affected by the refactoring.

By extending the object oriented language Java with implicit invocation with implicit announcement (abbreviated IIIA) these issues are tried to be corrected. This extension integrates join point types in Java, which serves as interfaces between classes and aspects. These interfaces reduce the coupling between the two components and therewith improve the modularity.

Structure

This thesis consists of two parts. Part 1 introduces the concept of IIIA and provides the theoretical information of this approach. The second part, Part 2, documents the implementation of the compiler which supports the approach of IIIA. Furthermore the approach is tested at several examples for evaluating the usage and usability of IIIA.

1 Scattering of code is the distribution of similar code in different modules, whereas code tangling is, when multiple concerns are implemented in one module.

Part 1
Types and Modularity for Implicit Invocation with Implicit Announcement

Types and Modularity for Implicit Invocation with Implicit Announcement

Friedrich Steimann

Lehrgebiet Programmiersysteme
Fakultät für Mathematik und Informatik
Fernuniversität in Hagen
D-58084 Hagen
steimann@acm.org

Thomas Pawlitzki

Lehrgebiet Programmiersysteme
Fakultät für Mathematik und Informatik
Fernuniversität in Hagen
D-58084 Hagen
Thomas.Pawlitzki@fernuni-hagen.de

Abstract

Implicit invocation is both an architectural style and a programming paradigm. Recently, aspect-oriented programming has popularized a special form of implicit invocation, namely implicit invocation with implicit announcement, as a possibility to separate concerns that lead to interwoven code if conventional programming techniques are used. However, as has been noted elsewhere, implicit announcement as currently implemented establishes strong implicit dependencies between components, which hampers independent software development and evolution. Inspired by how interfaces and exceptions are realized in Java, we present a type-based solution to this problem that integrates naturally with object-oriented programming, in particular with its subtyping and inheritance. Our presentation is informal, yet provides some empirical evidence for the viability of our approach.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Constructs and Features – abstract data types, polymorphism, control structures.

General Terms Design, Languages, Verification.

Keywords implicit invocation; event-driven programming; publish/subscribe; aspect-oriented programming; modularity; typing

1. Introduction

Garlan and Shaw have defined implicit invocation as an architectural style. They introduce it as follows:

> *The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with the event. When the event is announced the system itself invokes all of the procedures that have been registered for the event. Thus an event announcement implicitly causes the invocation of procedures in other modules.* [18, p. 9]

Implicit invocation is also a programming paradigm, better known as *event driven programming* (EDP) or *publish/subscribe* (P/S) [12]. Szyperski characterizes it as follows:

> *Firing an event is similar to calling a procedure or method. However, the target of the event is totally unknown to the source of the event and there can be multiple targets for a single fired event. Event firing is not normally expected to return any results. Firing events is done as a service to other objects, not to fulfil local needs. Event models can be seen as a generalization of notification mechanisms, such as the one introduced in the Observer design pattern.* [44 , p. 157]

Not surprisingly, implicit invocation has benefits and disadvantages:

> *One important benefit of implicit invocation is that it provides strong support for reuse. Any component can be introduced into a system simply by registering it for the events of that system. A second benefit is that implicit invocation eases system evolution* […]. *Components may be replaced by other components without affecting the interfaces of other components in the system.* […]

SIGPLAN'05 June 12–15, 2005, Location, State, Country.

The primary disadvantage of implicit invocation is that components relinquish control over the computation performed by the system. When a component announces an event, it has no idea what other components will respond to it. […] *Finally, reasoning about correctness can be problematic, since the meaning of a procedure that announces events will depend on the context of bindings in which it is invoked. This is in contrast to traditional reasoning about procedure calls, which need only consider a procedure's pre- and post-conditions when reasoning about an invocation of it.* [18, p. 10]

A special form of implicit invocation is *implicit invocation with implicit announcement* of events (hereafter abbreviated as IIIA), in which events are not published through a dedicated statement, but are instead specified declaratively. According to [17, 32], prominent applications of IIIA are database triggers and wrapper functions in CLOS; it is described by the authors as permitting "events to be announced as a side effect of calling a given procedure". This is considered "attractive because it permits events to be announced without changing the module that is causing the announcement to happen." [17, 32]. However, this gain also worsens the problems quoted above; in particular, it adds to the ignorance of a component, which under IIIA does not even know that it announces an event.

Aspect-oriented programming (AOP) [23] can be viewed as a contemporary form of IIIA [45]. Indeed, the most popular AOP language to date, AspectJ, has a powerful, declarative *pointcut language* that allows one to select from certain points of execution in a program, called *join points*, those with which an (implicitly announced) event is associated. By binding pointcut expressions to methods called *advice*, implicit invocation of these methods takes place whenever the corresponding pointcut fires (matches). The announcement of the corresponding event can therefore be considered implicit.

More recently, concerns have been raised that IIIA à la AOP is anti-modular in that it establishes a strong, implicit coupling between the components of a system [1, 6, 7, 19, 20, 33, 36, 41, 42, 43, 45]. Especially the absence of explicit interfaces, or other hints in the places where behaviour may get changed, is thought to hamper independent development. This is in stark contrast to the above quoted benefit of implicit invocation, namely the easing of system evolution.

In this paper, we present a simple solution to the problems of IIIA that restores full modularity of involved components. It evolved out of our own prior work on avoiding accidental recursion in AspectJ by introducing type levels [4, 15], and of our criticism of AOP including the solutions suggested in the literature to date [41]. Our approach is based on the novel concepts of *join point types* as the types of events that can be implicitly announced, and *polymorphic pointcuts* as their type predicates that are defined as parts of the classes exhibiting join points. Our solution, which we present as an AspectJ-based extension to the Java programming language, blends naturally with Java's native programming concepts; in particular, it bears some similarities with its type-based notions of interfaces and exception handling.

Table 1: Terminology, rough equivalences

Aspect-Oriented Programming, this paper	Event-Driven Programming, Publish/Subscribe
join point	event
advice	event handler
join point type	event type
`exhibits` declaration	`publishes` declaration
`advises` declaration	`subscribes` declaration
pointcut, join point type predicate	implicit announcement

The remainder of this paper is organized as follows. We begin with an introductory example in Section 2, which demonstrates the problem we are attacking and also builds the bridge from general IIIA to AOP. In Section 3 we introduce the basic concepts of our solution, namely join point types, polymorphic pointcuts, and explicit join point creation. Section 4 extends these concepts to subtyping and inheritance, taking the interaction with class hierarchies into account. The full language is presented in Section 5; however, its semantics is only informally sketched, by showing how it is translated to the constructs of AspectJ. Section 6 provides a short evaluation based on the experiments we have conducted. Discussion of related work and a conclusion complete our contribution.

One further remark before we begin. This paper is at the intersection of OOP, EDP (or P/S), and AOP. This imposes a terminological problem, namely which labels to use for the terms we rely on. After several full rewrites of the paper we decided to stick with the jargon of AOP, AspectJ in particular, mostly because the target language of our compiler is AspectJ and many of its concepts shine through. For readers unfamiliar with AOP and better acquainted with EDP, Table 1 may serve as a cheat sheet helping through the paper.

2. A motivating example

IIIA is perhaps best known in the world of relational databases: so-called *triggers* allow the interception of database operations and their enhancement with stored procedures [11]. To quote from the reference manual of MySQL 5.0:

A trigger is a named database object that is associated with a table, and that activates when a particular event occurs for the table. Some uses for triggers are to perform checks of values to be inserted into a table

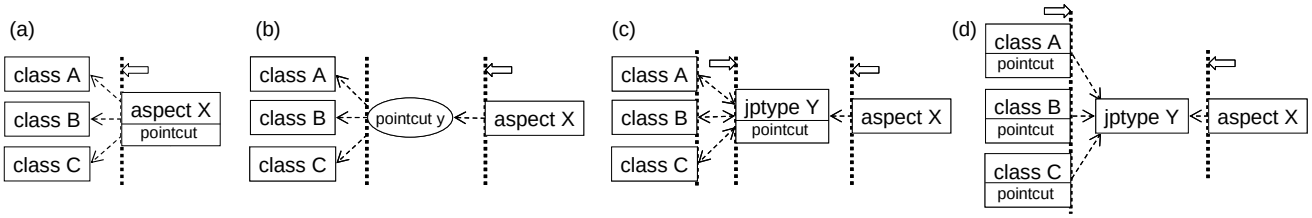

Figure 1. From standard aspects to typed and modular IIIA. (a) Aspect with local pointcut. (b) Same aspect with pointcut moved in proximity of targets. (c) Pointcut encapsulated by a join point type (jptype) and classes declaring to exhibit corresponding join points. (d) Pointcut split and branches moved into targets (polymorphic pointcut). Dashed arrows represent referencing and change dependency, vertical dotted bars represent interfaces, and hollow arrows the direction from which they are programmed against.

> *or to perform calculations on values involved in an update. … A trigger is associated with a table and is defined to activate when an INSERT, DELETE, or UPDATE statement for the table executes. A trigger can be set to activate either before or after the triggering statement. For example, you can have a trigger activate before each row that is deleted from a table or after each row that is updated.* [31]

For example, the following MySQL statement defines a trigger named `BonusProgram` on a table named `ShoppingCart`:

```
CREATE TRIGGER BonusProgram
BEFORE INSERT ON ShoppingCart
  -> FOR EACH ROW SET
     NEW.amount = NEW.amount + NEW.amount / 2;
```

It means that before records (called rows) are inserted into `ShoppingCart`, the procedure after the `->` is implicitly invoked, incrementing the value of the rows' `amount` field by one half. Readers familiar with AspectJ will note a certain similarity; indeed, what a trigger can do to database operations, an aspect of AspectJ can do to statements of a Java program.

To demonstrate this, we port our example to AspectJ and extend it slightly. First, we introduce a class `ShoppingSession` and three referenced classes `ShoppingCart`, `Invoice`, and `Log`[1]. The referenced classes all offer methods for adding an amount of items, the only difference being that `Invoice` takes a customer's personal rebate into account, which is why its `add` method receives the customer as an additional parameter.

```
package application;

class ShoppingSession {
 ShoppingCart s = new ShoppingCart();
 Invoice i = new Invoice();
 Log l = new Log("buys");
 Customer c = customerLogOn();

 void buy(Item item, int amount) {
  s.add(item, amount);
  i.add(item, amount, c);
  l.add(item, amount);
 }
}
class ShoppingCart {
 void add(Item item, int amount) {…}
}
class Invoice {
 void add(Item item, int amount, Customer c) {…}
}
class Log {
 void add(Item item, int amount) {…}
}
```

Then, after the application has been deployed, marketing wants to install a customer bonus program "buy 2 books, get 1 for free". Using AspectJ, this added behaviour can be realized by installing an aspect `BonusProgram`, which adapts the amount of books for all `add` transactions except that for `Invoice`:

```
package aspects;

aspect BonusProgram {

 pointcut buying(Item item, int amount):
   execution(* *.add(Item, int))
      && args(item, amount);

 void around (Item item, int amount):
      buying(item, amount) {
  if (item.category == Item.BOOK)
   amount += amount / 2;
  proceed(item, amount);
 }
}
```

The (named) pointcut `buying` specifies the condition that leads to the *implicit invocation* of the advice (the block introduced by `void around …`). Note that its specification is highly economical in that it specifies an open number of locations in the source code, a property sometimes referred to as *quantification* [14].

Except for quantification, the situation in AspectJ is rather similar to that in SQL. In particular, in both cases only `BonusProgram` contains hints that, and where or when, implicit invocation takes place. From a software engineering perspective, however, this poses a serious modularity problem: while `BonusProgram` implicitly specifies on what it depends (in case of SQL through the `BEFORE` clause, in case of AspectJ through the pointcut `buying` it defines), the targets `ShoppingCart` and `Log` contain no hints of this coupling, a property referred to as *oblivious-*

[1] Note that logging is *not* implemented as an aspect.

ness in [14]. In particular, the lack of an explicit interface on the side of the target means that whenever one wishes to change the implementation of that target, one does not know which interfaces to respect. This situation is shown in Figure 1 (a).

To illustrate this problem for the case of AspectJ, suppose that after installation of the `BonusProgram` aspect it is discovered that the log needs a customer entry (changes highlighted):

```
class Log {
 void add(Item item, int amount, Customer c)
   {…}
}

class ShoppingSession {
 …
 void buy(Item item, int amount) {
   …
   l.add(item, amount, c);
 }
}
```

This change breaks the `buying` pointcut from above, which no longer matches `Log`'s `add` method. Although this can be fixed by adapting the pointcut as in

```
pointcut buying(Item item, int amount):
  (execution(* *.add(Item, int)) &&
  args(item, amount)) ||
  (execution(* Log.add(Item, int, Customer))
  && args(item, amount, ..));
```

nothing in `Log` informs the programmer of this necessary change. The untoward effect this has on modularity and independent development has been discussed, e.g., in [1, 6, 7, 19, 20, 33, 36, 41, 42, 43, 45].

In the following, we will show how we have solved this problem for IIIA based on AspectJ. We expect that our solution can be transferred to other implementations of IIIA, yet make no definite claims in this regard. However, we note that the transfer to SQL is straightforward.

3. Join point types, polymorphic pointcuts, and modularity

Modularity problems such as the one just described are usually solved through the introduction of interfaces, i.e., "shared boundaries across which information is passed" [22]. In our example, boundaries are shared between classes `ShoppingCart` and `Log` on the one side and the aspect `BonusProgram` on the other, and the information passed consists of the parameters `Item item` and `int amount`, as well as the fact that the (type of) event that has triggered the implicit invocation has been named "buying" (through the pointcut with which the advice is associated). However, declaration of this interface remains implicit in `BonusProgram` (it can be derived from the pointcut `buying`), and is completely absent from the classes. Our first step to restore full modularity is to make the boundary and the information passed explicit, on both sides.

3.1 Join point types

Inspired by typed P/S [13] and also by Java's type-based exception handling, we interpret join points as typed events and introduce *join point types* as first class constructs that serve to specify the interface between classes exhibiting join points and aspects handling them. In the case of our example, we define the following join point type:

```
joinpointtype Buying {
  Item item;
  int amount;
  pointcut execution(* *.add(Item, int, ..))
   && args(item, amount, ..);
}
```

This type gets instantiated every time a join point covered by its pointcut occurs in the program. The instance's fields are bound to the parameters of the context in which the join point occurs, as prescribed by the pointcut. Because it characterizes the nature of its instances, we think of the pointcut as a *type predicate*.

Join point types like this let us declare interfaces (boundaries and information passed) between classes and aspects. In our example, we add the following clauses to make the interfaces explicit:

```
class ShoppingCart exhibits Buying {…}

class Logger exhibits Buying {…}

aspect BonusProgram advises Buying {…}
```

The `exhibits` clauses mark the *caller* side of implicit invocation, and the `advises` clause the *called*. This may appear counter to intuition, since the aspect (as the "advisor") seems to be the active, and the class (the "advised") the passive, and indeed the aspect depends on the classes it advises and not vice versa; however, such reversal of dependency is not unusual for interfaces (so-called *enabling interfaces* [40]). The situation is depicted in Figure 1 (c)

Definition of the join point type `Buying` as above allows us to rewrite the aspect `BonusProgram` as follows (changes highlighted):

```
aspect BonusProgram advises Buying {
  before (Buying jp) {
    if (jp.item.category == Item.BOOK)
      jp.amount += jp.amount / 2;
  }
}
```

The advice is now parameterized by the variable `jp` of type `Buying`, which holds the join point instance that led to the implicit invocation of the advice, and which (through its fields) provides access to the context in which the join point occurred.[2] Note that we have changed `around` to `be-`

[2] What seems like a cosmetic change can have far-reaching consequences: by assigning the join point instance to another variable, it need not be handled immediately, but like an event can be stored in a queue for later, asynchronous treatment, or like an exception can be re-exhibited (see Section 3.3 on how this can be done). However, because of the volatility of join points (which after all are points in the execution of a program), asynchronous write access to the context where a join point occurred may be

`fore` and have dropped the `proceed`: values written to the fields of a join point are written back to the actual parameters bound to the fields during join point creation, once the advice has completed. Following the usual convention of Java, writing to the fields can be prevented by declaring them final in the join point type.

Our use of join point types improves modularity in that maintainers of a class wishing to make changes to it can consult the definitions of the join point types the class exhibits, and observe the pointcuts specified there. However, two problems remain.

1. Pointcuts are currently still purely syntactical constructs. This means that while it is clear that refactoring of a class should make sure that all pointcuts of the exhibited types produce the exact same set of join points after the refactoring, it is not so clear how to deal with semantic changes: should a new method match a given join point, should an existing method be changed to match or no longer match one? For this, a semantic specification of the pointcuts would be necessary.[3]
2. In practice, the surface structure (appearance) of join points of a single join point type can vary greatly from class to class, in AspectJ typically resulting in complex pointcuts consisting of many disjuncts (the "*quantification failure*" noted in [43]). Such a pointcut mirrors the structure of the classes it advises, and changes in this structure require changes in the pointcut, again compromising independent development.

Regarding the first problem: because languages for formally specifying "semantic pointcuts" [28, 35] (analogous to design-by-contract languages [29] that can specify the semantics of an interface; cf. Footnote 3) are still mostly dreams of the future, we resort to an informal description of the nature of the join points covered by a join point type (as has also been done for the crosscutting interfaces described in [19, 43]; cf. related work for a discussion). It is then the responsibility of the developer of each class exhibiting such a join point type that the join points matched by the pointcut conform to this informal specification. Regarding the second problem, we can offer a much more tangible solution.

undefined for temporary variables (actual parameters and temporaries), so that access should be limited to read only (see the discussion of spectators and assistants in Section 7). We therefore decided that allowing explicit assignment of join points may not be worth its price, and currently forbid it in our language extension.

[3] As an aside, note that Java's interfaces are also abstract type specifications consisting only of a set of method signatures. An implementing class can give such a type any meaning it chooses to, and different classes can give it different meanings. The meaning of the (abstract) type is then simply the union of all its implementations. This may appear unacceptable in some situations, and natural in others (e.g., implementation of the Runnable interface poses no semantic requirements).

3.2 Polymorphic pointcuts

Because classes already specify whether they exhibit join points of a certain type (through a corresponding `exhibits` clause), it seems only natural to let the classes themselves specify the (part of the) join point type predicate (the pointcut) under which their join points fall. Transferred to our example, this means that the pointcut definition in `Buying` is dropped and the classes `ShoppingCart` and `Log` are extended as follows:

```
class ShoppingCart exhibits Buying {
 Buying pointcut:
   execution(* add(Item,int))
   && args(item, amount);
 …
}
class Log exhibits Buying {
 Buying pointcut:
  execution(* add(Item, int, ..))
  && args(item, amount, ..);
 …
}
```

The resulting dependencies are shown in Figure 1 (d). Note that the polymorphic pointcuts lack class information; the scope of each such pointcut is implicitly constrained to the class in which its definition occurs. The disjunction of all class-local pointcuts associated with a join point type then constitutes the complete pointcut of that type. Because this is reminiscent of how different classes implementing the same interface provide for polymorphic methods in Java (cf. Footnote 3), we call such pointcuts *polymorphic*. It is the responsibility of the programmer to make sure that polymorphic pointcuts conform to the informal specification associated with the corresponding join point type.

3.3 Explicit announcement of join points

Our view of join points as instances of types opens up an interesting opportunity: it allows us to announce join points explicitly, corresponding to the explicit creation of events. For instance, suppose that we want to add a counter cumulating the total number of items delivered, and that we therefore extend `ShoppingSession` and its method `buy` as follows:

```
class ShoppingSession {
 int totalAmount = 0;
 …
 void buy(Item item, int amount) {
  s.add(item, amount);
  i.add(item, amount, c);
  l.add(item, amount, c);
  totalAmount += amount;
 }
}
```

The added statement must be advised by `BonusProgram` to maintain consistency, but because this statement does not involve the variable `item` (access to which is needed by the advice), formulation of a suitable pointcut is unobvious — a problem reported to be not infrequent in practical applications of AspectJ (so called *state-point separation* and *inaccessible join points* [43]). Rather than rewriting our pro-

gram so as to allow pointcut matching (the *intimacy* described in [9, 43, 45]), we introduce the following construct that creates the join point instance with all required parameters explicitly:

```
class ShoppingSession exhibits Buying {
  void buy(Item item, int amount) {
    …
    exhibit new Buying(item, amount) {
      totalAmount += amount;
    };
  }
}
```

The type of this newly created join point, which does typically not fall under the type predicate (pointcut) of its declared type `Buying` (because otherwise the explicit creation would be redundant), can be thought of as an anonymous inner subtype (analogous to the anonymous inner classes of Java), i.e., as a join point subtype that comes with its own, implicit type predicate. More on subtyping in Section 4.

3.4 Modularity achieved

Letting a class declare that it exhibits join points of a certain type expresses a *statement of consent* that some of the classes' variables may be accessed by aspects advising the exhibited join points. Moreover, the local pointcut specification specifies within the class which of its variables can get accessed. Conversely, it gives classes the opportunity to deny aspects access; in particular, and much in the spirit of information hiding, variables can only be accessed if the owning class explicitly grants access to them.

As a consequence, our proposal makes evolution of classes completely independent from aspects: anyone wishing to make changes to the class can check locally, without resorting to any other declaration or definition, whether one's changes respect the advising aspects' interfaces. Much more: *in case one must break with a (local) pointcut definition, one can adapt it without affecting the definitions in other classes, because the scope of a local branch of a join point is always limited to the owning class*. In fact, the only thing the programmer must guarantee is that the variables declared in the join point type (e.g., `Item item` and `int amount` of `Buying`) are correctly bound to variables in the context of the local join points. If that is impossible using a (local) pointcut definition, one can still work around it with the explicit creation of (an instance of) a join point (via the `exhibit` clause as shown in Section 3.3). This means that the advised classes can be changed at will, as long as the local pointcut can be adapted accordingly.

To address concerns that local pointcut definitions limit the expressiveness of our language proposal unduly, we could offer the combination of "crosscutting", global pointcut definitions with local pointcuts, by allowing a global pointcut definition to be overridden locally. More specifically, we could allow a join point type declaration (such as that of `Buying` in Section 3.1) to include a pointcut definition that is inherited by the classes exhibiting this join point type, while at the same time admitting that this (global) definition is overridden by a local one. However, as the next section will show, the same effect can be achieved by making an abstract class exhibit a join point type, and letting the its subclasses inherit the pointcut defined. The impatient reader can jump to Figure 6 for an example of this.

4. Subtyping and inheritance

Experience with object-oriented programming languages has taught that subtyping and inheritance are sources of considerable (and also often unexpected) complexity. In order not to repeat errors of the past, we attempt a systematic analysis of what subtyping and inheritance mean for join point types, and how this combines with subtyping and inheritance of classes. For this purpose, we look separately at *intensions* and *extensions*, i.e., the definitions of types and the sets of objects falling under these definitions.

4.1 Subtyping and inheritance for join point types

If we want to treat join point types as types on the same footing as classes, interfaces, and exceptions, subtyping means that instances of a join point subtype can occur anywhere instances of its supertypes are required (extensional view) [27]. This is naturally the case if whatever holds for (the instances of) a join point supertype, also holds for (the instances of) its subtypes (intensional view). This in turn is granted by inheritance, i.e., by the propagation of properties from a supertype to its subtypes, as long as subtypes do not override the inherited properties in an incompatible way.

4.1.1 Extensional view

Recursively dividing the set of all join points of a program into subsets, and viewing each such subset as the extension of a corresponding join point subtype, seems natural. As Figure 2 suggests, this division is independent of, and may be orthogonal to, the association of join points with the classes and subclasses hosting them.[4] Orthogonality, if present, reflects the crosscutting property commonly associated with aspects: it indicates that sets of join points regarded as being related (as expressed by belonging to the same type) cannot be assigned to a single branch of the class hierarchy, but rather crosscut it.

[4] For the sake of simpler presentation, we have equated join points with join point shadows [21], that is, with locations in the source code whose execution may result in an implicit invocation. Every such join point shadow may represent a potentially infinite set of join points.

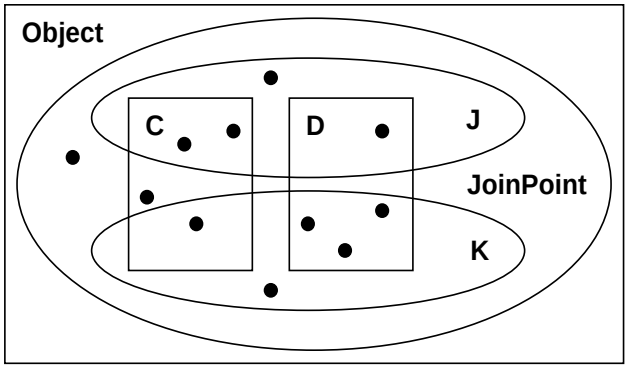

Figure 2. Class `Object` and two subclasses `C` and `D` (all depicted as rectangles), all possessing join points (depicted as dots). Join points are also classified by join point types, here `JoinPoint`, `K`, and `J` (depicted as ellipses), which form an independent type hierarchy.

Note that orthogonality is not a necessary condition for join point types. Yet there is a fundamental difference between the partitioning of the set of join points of a program through classes and through join point types: while every join point belongs to precisely one class (so that superclasses do not include the join points of their subclasses), the extension of a join point type includes the extensions of its subtypes.

Set inclusion semantics of join point subtyping dictates that a join point handler (advice) that accepts join points of a certain type must also accept join points of its subtypes. As we will see below, the requirements for this substitutability, namely the availability of the fields that are declared for the join point supertype, are naturally met. Conversely, a single join point of a certain type can be handled by various handlers, namely by those for its type and those for its supertypes. Since join points lead to (implicit) invocation with the join point as a parameter[5], we follow the generally established rule of method binding in presence of subtyping, namely that the most specific handler accepting the join point is invoked (cf. the discussion of polymorphism in AspectJ in the related work). Note that this handler is determined per aspect, i.e., in every aspects that has a matching handler, the most specific one is invoked. This maintains the broadcast semantics of implicit invocation.

4.1.2 Intensional view

The intension of a join point type consists of its field declarations as well as an (informal; cf. above) specification of the nature of its instances. This intension is inherited to its subtypes, meaning that an instance of a join point subtype has the same fields and must obey the same (informal) specification as those of its supertypes. Subtypes may however add to the intension: they can add fields and strengthen the specification. This ensures that a join point handler (advice) for a supertype can also accept join points of its subtypes).

Of particular interest is the case in which a single class exhibits both a join point type and one of its subtypes: the pointcut specifications must then be consistent in that the pointcut of the subtype must imply that of its supertype (so that the supertype's extension includes that of its subtype). Analogous to Eiffel's inheritance of assertions [29 , Sect. 16.1], this could be ensured by implicitly disjoining pointcuts defined in a class. However, such is not needed in practice, since a join point of a certain type is automatically treated as if it were one of its supertype, should a specific handler be missing (cf. above). On the other hand, what must be guaranteed is that a class does not create two join points for one executed statement (cf. [10] for how this is currently the case in AspectJ), namely one for the join point type and one for its supertype. To make sure that only the most specific pointcut defined in a class creates a join point, we automatically conjoin the pointcuts of join point supertype with the negations of the pointcuts of their subtypes specified in the same class.

4.2 Effect on subtyping and inheritance for classes exhibiting join points

An entirely different, yet no less important issue is whether subclasses exhibiting join points are proper subtypes of their superclasses. Basically, this would require substitutability, i.e., the fact that instances of the subclasses, with the added behaviour from advice, behave in the same manner as those of its superclasses (which may also, but need not, be advised by same or different aspects). Whether this is the case is nontrivial to decide (and has been classified largely as an open problem [6]); however, we will argue that the situation for IIIA is no worse than for (late bound) method calling.

4.2.1 Extensional view

The extensional view boils down to the question of whether an object of a subclass that exhibits certain join point types can occur anywhere in a program where an object of its superclass, which may exhibit other or no join point types, is expected. Syntactically, the answer is yes, since IIIA does not change the protocol of a class — it only adds implicit method invocations. Semantically, however, adding method invocations is similar to overriding, as it

[5] In fact, implicit invocation could be thought of as being dispatched on the join point instance as the receiver, which would make aspects behaviour-delivering implementors of join point types. However, we do not pursue this interpretation further here (but cf. Related work).

can change behaviour.[6] Is the class with the changed behaviour still a proper subtype of its superclass?

To answer this question, one could attempt to set up a formal framework that links advice to classes, and formulate formal conditions that must be met by advice affecting the code of subclasses. This could, e.g., include rules of covariance (or contravariance?) of exhibited join point types. On the other hand, since subclasses may add code that can require advice not anticipated by the superclass, such rules are not easily identified. For this, we decided to retreat to the general position that IIIA must not lead to a behaviour that breaks the contract of a (super)class. As long as all join points reside within methods, pre- and post-conditions as demanded by design-by-contract [29] should be sufficient to check substitutability of classes at runtime. Verifying it statically in a modular fashion (i.e., without resorting to a whole-program analysis) seems possible at least under the same conditions as for implicit invocation with explicit announcement (since advice is linked to a class via `exhibits` and `advises` clauses, and the pointcuts of a class can be locally translated to explicit calls; see also [8]); however, we do not pursue this further here.

4.2.2 Intensional view

Regarding inheritance, the question is whether (class local) pointcuts and corresponding `exhibits` clauses should be viewed as parts of the intension of a class and as such be inherited to its subclasses. As argued above, that a class hosts join points does not imply that its subclasses also do. In fact, even though a class whose behaviour is changed through the exhibition of join point types (see above) inherits this (changed) behaviour to all subtypes, it is still only the class, and not its subclasses, that actually exhibit the triggering join points. Also, a pointcut specified in a class is a local branch of the type predicate associated with the corresponding join point type and as such a part of the join point type's intension, not the class's. In particular, it has to satisfy the (informal) specification of the join point type, not that of the class. It is related to the class only insofar as it maps the specification of the join point type to the implementation of the class (the occurrence of join points in the code of the class).

On the other hand, that a subclass does not inherit join point exhibition and pointcuts from its superclasses does not mean that it cannot exhibit the same kind of join points — indeed, this may make sense in certain situations. The question, then, is whether `exhibits` and corresponding pointcuts should be inherited by default and cancelled if desired, or whether they should not be inherited, but may be explicitly declared (reintroduced) by a subclass, which

```
type_specifier ::=
  …
  | joinpoint_name
joinpoint_name ::=
  identifier | ( package_name "." identifier )
type_declaration ::=
  …
  | joinpoint_declaration
joinpoint_declaration ::=
  "joinpointtype" identifier
  "{" { joinpoint_field_declaration } "}"
joinpoint_field_declaration ::=
  [ "final" ] type identifier ";"
class_declaration ::=
  { modifier } "class" identifier
  …
  [ "exhibits" joinpoint_name [ { "," joinpoint_name } ]
]
  "{" { class_body } "}"
class_body ::=  { class_member }
class_member ::=
  …
  | joinpoint_pointcut_declaration
block ::=
  …
  | exhibit_block
exhibit_block ::=
  "exhibit new " joinpoint_name "(" [ {argument } ] ")"
  "{" { statement } "}"
joinpoint_pointcut_declaration ::=
  "pointcut" joinpoint_name ":" pointcut_expression ";"
pointcut_expression ::=
  …
  | "super"
aspect_declaration ::=
  { modifier } "aspect" aspect_name
  [ "advises" joinpoint_name [ { "," joinpoint_name } ] ]
  "{" { joinpoint_advice_declaration } "}"

joinpoint_advice_declaration ::=
  ("before"|"around"|"after")
  "(" joinpoint_name variable_declarator ")
  {" advice_content "}"
```

Figure 3. Syntax of our language extension.

then may refer to the superclass's pointcut using the `super` keyword.[7] Because inheritance is generally known as a problem for modularity [30], and because modularity is one of our main goals, we opt for the latter, establishing a clear inheritance interface with respect to join point exhibition between a subclass and its superclasses. In particular, this lets the implementer of a subclass be always aware and in control of the exhibited join points, avoiding the fragile base class problem for join points [42].

5. The full language, finally

With the subtyping-related issues clarified as above, we are now ready to complete the specification of our language.

5.1 Syntax

The syntax rules that add our language extension to the AspectJ grammar as specified in the AspectBench Compiler [2] is shown in Figure 3. According to this grammar, the program shown in Figure 4 is syntactically correct.

[6] Note that for a change of behaviour, direct write access to the context of join points is not needed; the triggered advice can change the state of the exhibitor via its public interface.

[7] Note that pointcuts inherited this way are restricted to the scope of the subclass. Among other reasons, this avoids that a subclass can publish join points of its superclasses, which the superclasses did not publish.

```
joinpointtype J {…}
joinpointtype K extends J {…}
joinpointtype L extends K {…}

class C exhibits J {
  pointcut J : …
  …
}

class D extends C exhibits J, K {
  pointcut J : super;
  pointcut K : …
  pointcut L : … // semantic error
  …
}

class E extends C exhibits L {
  pointcut L : …
  …
    exhibit new L() {…}
}

aspect X advises J, K, L {
  before (J j) {…}
  after (K k) {…}
}
```

Figure 4. Sample code highlighting language features.

5.2 Semantics

We sketch the semantics of our language by providing the mapping of its constructs to those of AspectJ. The mapping is as follows:

- A join point type maps to a class with the type's fields and a constructor for creating instances and setting the fields; a join point subtype maps to a corresponding subclass.
- A class local pointcut maps to a correspondingly restricted disjunct of a global pointcut.
- An aspect maps to an aspect; an advice linked to a join point type maps to advice bound to the corresponding global pointcut created from the class-local branches.
- To emulate subtyping of join point types, `advises` clauses for which no specific advice exists map to additional advice in aspects providing advice for the supertypes only (cf. Section 4.1.2).
- Finally, explicit announcement of a pointcut maps to a specially tagged block (see below).

Note that except for the subtyping mentioned above, `exhibts` and `advises` clauses are used for semantic checking only; they are compiled away. Figure 5 shows the result of this translation when applied to the code of Figure 4.

5.3 Implementation of the compiler

We have implemented a compiler for our form of IIIA on top of the AspectBench Compiler (abc) [2]. It adds a number of compiler passes, which are roughly characterized as follows (passes performing semantic checks omitted):

```
class J { J(…) {…} …}
class K extends J { K(…) {…} …}
class L extends K { L(…) {…} …}

class C {…}
class D extends C {…}
class E extends C {
  …
  {…}
  …
}

aspect X {
  before <pointcut for J>: {… proceed(…);}
  before <pointcut for K>: // same as above J
  before <pointcut for L>: // same as above J
  after <pointcut for K>: {…}
  after <pointcut for L>: // same as for K
}
```

Figure 5. Code of Figure 4 translated to standard AspectJ

1. The first pass collects all join point types and creates a new node holding the fields, a constructor setting the fields (including those inherited from supertypes), and an empty pointcut definition for each type.
2. The second pass visits all classes, collects all pointcut branches specified in each class, explicitly restricts the scope of each branch to the class in which it occurred (by adding a corresponding `within` clause), adds an exclusion clause in case the same class specifies also pointcuts for join point subtypes (cf. Section 4.1.2), and adds it so-modified as a disjunct to the pointcut of the corresponding join point type. Explicit (ad hoc) join point creation is handled by adding an always-match tag to the designated statement, which directs abc's matcher to insert an unconditional (unguarded) advice invocation.
3. The third pass visits all aspects and binds each of its advices to the corresponding pointcut constructed in the second pass. It inserts a constructor call for the join point type at the beginning of each advice, which binds the pointcut parameters to the fields of the corresponding join point type. It also creates copies of the advice for all join point types that are subtypes of the types already advised by the aspect, and for which no specific advice is defined in the aspect (in order to mimic the subtyping of join point types; see Section 4.1.1). Finally, it adds the fields of the join point type to the `proceed` statement in around advice, and translates before advice to around advice with a proceed at the end.

The compiler together with additional material can be downloaded from www.fernuni-hagen.de/ps/prjs/IIIA/.

5.4 Summary of properties, and comparison with interface and exception types

In brief, our AspectJ-based extension of Java with IIIA has the following properties:

- It interprets join points as runtime instances of user-declared join point types, with fields of join point types binding to the context of a join point instance.
- It interprets pointcuts as type predicates of join point types.
- It requires that classes exhibit join points explicitly, as declared by an `exhibits <join point type>` clause.
- It requires the polymorphic definition of pointcuts by making classes declaring to exhibit join points of a certain type define their branch of the corresponding pointcut (type predicate) locally, the complete pointcut thus being defined as a disjunction of it class-local branches.
- It allows the explicit creation of join points at runtime via an `exhibit new <join point type constructor> {<statement>}` expression in all cases in which a suitable type predicate is difficult or impossible to formulate using the given pointcut language (not infrequent according to [43]).
- It makes the dependencies of aspects explicit, by requiring them to declare through an `adivses <join point type>` clause instances of which join point types they intend to advise.

Thus, join point types are like Java interfaces (analogies in parentheses)

- in that they are abstract, i.e., provide no instances of their own, but must recruit them from the exhibiting (implementing) classes;
- in that they specify *what* the exhibiting (implementing) classes must provide, namely the values of the fields that are declared in the join point type, while leaving the *how* to the classes; and
- in that they allow the creation of anonymous inner join point types (anonymous inner classes) via `exhibit new <join point type constructor> {<statement>}`.

As a result, each class can define the sets of join points it exhibits individually by providing its own type predicate, and the extension of each join point type is the union of the sets of join points of that type as specified by each class. This is roughly the same with interfaces, which let classes define the method implementations individually, and whose extension is the union of those of its implementing classes.

At the same time, *join point types are like exceptions*

- in that their instances may either come into existence at some implicitly specified point of program execution within the lexical scope of the `exhibits (throws)` clause, or are explicitly created with an `exhibit new <join point type constructor> {<statement>}` (`throw new <exception constructor>`) expression; and
- in that their occurrence is handled in some place remote from, and unknown to, where they occurred.

```
joinpointtype UpdateSignaling {
  // change of state that affects display
}

abstract class Shape exhibits UpdateSignaling {
  pointcut UpdateSignaling :
    execution(void moveBy(int, int));
  public abstract void moveBy(int dx, int dy);
}

class Point extends Shape
            exhibits UpdateSignaling {
  pointcut UdpateSignaling :
    super || execution(void set*(int));
  int x, y;
  public int getX() { return x; }
  public int getY() { return y; }
  public void setX(int x) { this.x = x; }
  public void setY(int y) { this.y = y; }
  public void moveBy(int dx, int dy) {
    x += dx;
    y += dy;
  }
}

class Line extends Shape
            exhibits UpdateSignaling {
  pointcut UdpateSignaling : super;
  private Point p1, p2;
  public Point getP1() { return p1; }
  public Point getP2() { return p2; }
  public void moveBy(int dx, int dy) {
    p1.x += dx; p1.y += dy;
    p2.x += dx; p2.y += dy;
  }
}

aspect Display advises UpdateSignaling {
  after (UpdateSignaling us) { update(); }
  static void update() {...}
}
```

Figure 6. Drawing example with polymorphic pointcuts.

Most strikingly, our dealing with join points resembles dealing with exceptions in that it avoids code tangling, but not scattering — each scope in which a join point may occur must be explicitly marked with the corresponding join point type. Therefore, our proposal is not useful for vastly crosscutting concerns such as logging or tracing.

6. Evaluation

We have applied our language extension to a number of publicly available AspectJ programs. The results were not surprising: pointcut definitions became smaller, but were distributed among classes, and the code became better readable, because the advising of classes was apparent from the classes themselves. To give an impression of this, Figure 6 shows our transcription of one of the standard AspectJ examples, used in [24] and elsewhere. Note its use of pointcut inheritance, while at the same time both obliviousness and quantification are reduced; yet, they are not

```
joinpointtype ItemProducing {}
joinpointtype ConsumerCreation {
  Consumer consumer;
}

class Producer exhibits ItemProducing {
  pointcut ItemProducing :
        execution(Item produceItem());
  …
}

class Consumer exhibits ConsumerCreation {
  pointcut ConsumerCreation :
        execution(new(..)) && this(consumer);
  …
}

class Item {…}

aspect Dispatcher
      advises ItemProducing, ConsumerCreation {
  List consumer = new ArrayList();
  after(ConsumerCreation creation) returning {
    consumer.add(creation.consumer);
  }
  after(ItemProducing producing)
                        returning(Item it) {
    // dispatch item to consumer
  }
}
```

Figure 7. Event-driven consumer/producer communication based on IIIA. Note how both parties remain completely unaware of each other, and also of the dispatcher.

removed entirely, but restricted to single classes. The remaining, intra-class obliviousness can be eliminated by tool support making join point shadows visible in the source code (as already available for Eclipse's AspectJ plugin). Since these pointcuts are specified within the class and are under exclusive control by the developers of the class, modularity is guaranteed. Figure 7 provides another popular example, this time with two different join point types and a single aspect handling events of both types. Last but not least, we are confident that we can solve most problems of *state-point separation*, *inaccessible join points*, and *quantification failure* described in [43] through explicit join point creation; however, since we only transformed existing AspectJ programs (which all had more or less suitable pointcuts), we have collected only little empirical evidence in this regard.

One question that remains, though, is whether IIIA in the form made possible by our extension of Java will add real value in practice. By now it should be clear that because of its strongly restricted forms of obliviousness and quantification, and also the lack of inter-type declarations, it cannot generally replace for AspectJ or other forms of AOP. In particular, many of the standard applications of AspectJ, including the usual logging, tracing, but also the implementation of certain design patterns (those requiring introductions), are not reasonably expressed using our form of IIIA. On the other hand, as demonstrated in the examples of Figure 6 and Figure 7, it can be used for EDP. EDP is certainly successful, at least as judged by the frequency of occurrence of the Event Notification [39] and Observer [16] patterns in current software systems. EDP's usefulness even in non-distributed systems is also evidenced by the fact that it is slowly beginning to move from the pattern status to a native construct in major programming languages — for instance, C# offers some basic support, via delegates and constructs for registering them with a publisher[8]. Evidence for the usefulness of implicit announcement is harder to find; knowledge and use of database triggers is mostly confined to the database community; yet the implementation of business rules in Java as described in [25], like our example from Section 2, can be based on implicit announcement. The added value of our approach is that it makes it modular.

7. Related work

Event-driven programming and publish/subscribe In EDP, registering and unregistering of subscribers usually occur at runtime, whereas in our approach to IIIA they are "woven in" using the weaving mechanism of an aspect-oriented programming language (see, e.g., [3, 21], but also [38] for a viable alternative). Also, the announcement, or firing, of events in EDP is usually explicit, while it is by definition implicit in our approach (there is no *publish* statement or explicit call of a corresponding procedure). Types have been introduced to EDP and P/S mainly as filters for subscribers [13]: rather than accepting every event and checking it individually for relevance, a subscriber subscribes only to certain types of events. By contrast, we use types mostly to specify interfaces on the side of the publisher, a purpose that is explicitly declined by proponents of implicit invocation [17, 32]. Denying interfaces sacrifices modularity, which we restore.

Aspect-oriented programming According to most common definitions of AOP, what we suggest is no longer aspect-oriented. For instance, compared to AspectJ it does not perform well in the removal of scattered code, and therefore does not modularize crosscutting concerns in the way expected by many in the aspect community. Compared to symmetric approaches such as Hyper/J [34], it performs even worse — because of its restriction to invocation, it does not support the merging of classes (or aspects) delivering aspect-wise structure and behaviour (introduction of new features into classes is not supported).

It follows that implementing important standard aspects such as logging or tracing, and also certain design patterns requiring structural introductions, with our proposal is no good idea. Also, in terms of the much-cited quantification

[8] Smalltalk, which may be viewed as the origin of the Observer pattern, already had it implemented as one of its control structures.

and obliviousness characterization [14], our proposal does not make it as a form of AOP: quantification is restricted to classes declaring to exhibit join points, and obliviousness is compromised to the extent that all classes in which join points may occur must be explicitly tagged as such. In fact, we even go as far as permitting explicit marking of individual join points through the `exhibit new <join point type>` construct, which eliminates obliviousness and quantification completely.

Adding Polymorphism to AspectJ Ernst and Lorenz noted that the polymorphism present in AspectJ is basically ad-hoc; all available inclusion polymorphism is that of the base language (Java) [10]. In order to introduce late binding of advice, the authors require some kind of advice grouping, so that a binding algorithm can "choose exactly one most specific advice and invoke it, ignoring all the others in the group (they are being *overridden*)." [10]. By our introduction of join point types and subtypes, and by linking advice to join point types (providing some kind of "advice signatures" [10]), we have installed such groupings. However, with the language and its translation to AspectJ as defined above, advice is still bound at runtime; in particular, we have not yet explored whether and how our approach could open the door for separate compilation.

Reduction of AOP to implicit invocation Xu et al. have shown how aspect-oriented programs can be *automatically* reduced to implicit invocation, so that available model checking approaches designed for implicit invocation can be used for aspect-orientated programs also [44]. However, as the authors themselves admit, the practicality of their approach is limited by the practicality of model checking in general: formulation of conditions to be checked is difficult, scalability is poor, and translation of the results (found counterexamples) back to the original input, in this case aspect-oriented programs, is nontrivial. By contrast, we have suggested an intuitive and simple to use type system that lets the compiler make certain checks, and sketched how semantic conditions can be ensured both dynamically and statically, using the traditional means of program verification.

Type-theoretic interpretation of pointcuts and advice In [26], Ligatti, Walker, and Zdancewic present formal semantics for an idealized AOPL. For this, they extend the simply-typed lambda calculus with two new abstractions covering join points, pointcuts, and advice, and prove type safety for this calculus. They present a small functional language, MiniAML, and show how this maps to the core calculus. MiniAML has some similarities with our language, most prominently that it allows scoping of advice: functions can be hidden from advice, thereby allowing "programmers to retain some control over basic information hiding and modularity principles in the presence of aspects." [26] The mapping of MiniAML to the core calculus is non-trivial; we expect a corresponding mapping of our own language, although certainly desirable to prove the soundness of our type system, to be no easier. On the other hand, what we have delivered can be immediately tried out in practical settings, allowing the community to improve it until it is maximally useful.

Stratified aspects In our own previous work, we proposed [15] and implemented [4] an extension of AspectJ that adds *type levels* to its join points and aspects. In the resulting language, type information in a program is partly implicit, and for the rest consists of `meta` modifiers attached to aspects and pointcuts. According to this type system, all join points contained in classes are of type level 0, all in aspects of type level 1, all in aspects declared with a single meta modifier of type level 2 and so forth. Pointcuts to range over join points of type level 0 remain unmodified, while those to range over type level 1 and higher have to be modified with a corresponding number of `meta` modifiers. This allows us to build towers of aspects as advertised in [37], albeit on the class rather than the instance level (cf. below). As can easily be seen, our current type system can emulate our previous one, simply by dividing the set of join point types into disjoint subsets each associated with a type level, and requiring that aspects advise only join point types from levels lower than the join points they themselves exhibit (if that is what they do; aspects exhibiting join points are not discussed in this paper). In fact, it should even be possible to automatically construct the type strata from the `exhibits`/`advises` relationships found in a program, and to report a typing error (or warning) should the relationship contain circles (potentially leading to self-application and recursion).

Classpects and Eos-U To achieve greater conceptual integrity, the "classpects" of Eos-U [37] drop the distinction between classes and aspects and let instances advise other instances. However, this requires binding of advising to advised objects, which introduces additional dependencies. By contrast, we have introduced join point instances that are automatically created when a pointcut matches, and let advice operate on these instances as if it were a method of the corresponding join point type (cf. Footnote 5). Our gain in conceptual integrity is therefore comparable. On the other hand, we believe that Eos-U would profit from the typing we suggested: for instance, its `addObject` method could be typed to accept only objects exhibiting the advised join point type.

Crosscutting Interfaces (XPIs) Griswold et al. suggest the introduction of crosscutting interfaces (XPIs) as interfaces "that base code designers 'implement' and that aspects may depend upon" [19]. For this, each XPI comes with a "syntactic part" that exposes the signature of named

pointcuts, and a "hidden implementation" [19, p. 54], the part that specifies the concrete pointcut expressions. XPIs are enhanced by informal, "semantic" specifications of join points that need to be observed by the implementers of classes. Note that storing the implementation, the pointcuts, in the interface is somewhat unusual (cf. the discussion of attaching pointcuts to join point types in Sections 3.1 and 3.2), but must be seen as technical tribute to AspectJ as the language in which XPIs are currently implemented. However, this technicality impairs independent module evolution to a certain extent, since the implementation of the interface is not part of the implementation of the module (so that decoupling reaches only stage (b) in Figure 1). Adopting our language extension would let XPIs achieve full modularity (stage (d) in Figure 1), by letting interfaces be implemented polymorphically, that is, per implementing class. At the same time, it addresses elegantly many of the problematic issues of AspectJ identified in [43], which are mostly due to the inability to formulate pointcuts that readily match the intended points in a program.

Open Modules Following Aldrich's influential work [1], Ongkingco et al. [33] present an implementation of Open Modules for AspectJ. It introduces a module concept as an owning collection of classes that together declare a set of friend aspects (that can freely access all classes of the module) as well as specific pointcuts advertised or exposed (the difference is of little importance here) to aspects. All join points included in the module that are not exposed are invisible from the outside. In addition, a module may expose join points selectively to aspects that it names. This is somewhat comparable to, although still sufficiently different from, our approach in which join points are specifically exposed to classes that declare to depend on the join points' type. In sharp contrast to our work, however, is that in Open Modules classes remain unaware of the join points they expose, and also of the pointcuts specifying those join points. In particular, in Open Modules à la [33] the pointcuts used by an aspect cannot be adapted and maintained on a per class basis, thereby limiting independent evolution of aspects and base classes to a certain extent. Also, the ability to declare friend aspects of a module, while allowing such things as debugging via aspects, provides for unspecified (implicit) interfaces to the module, which basically implies that friend aspects are part of the modules whose classes they advise.

Spectators and assistants Clifton and Leavens use the `accept` keyword to let classes declare that they admit advice from the aspects listed thereafter [5]. In Figure 1, this would correspond to (a) with bidirectional dependencies. We are taking a different route: by introducing join point types as middle men between aspects and their targets, and by introducing class-wise polymorphic pointcuts, we reach the degree of decoupling shown in Figure 1 (d). Clifton and Leavens [5, 6] further distinguish between spectators (aspects that may only observe) and assistants (aspects that can actually change state). Using our approach, and would Java offer a modifier similar to C++'s `const`, we could allow declaring single fields of a join point type as being observable only, or as being changeable, thereby granting finer-grained access control. On the other hand, preventing direct write access to objects cannot prevent the behaviour changing interception of methods. For a more detailed discussion of how potentially interfering aspects can be separated from "harmless advice", we refer the reader to [7].

Aspect-implied interfaces In their effort to restore modularity of AOP, Kiczales and Mezini argue that "aspects cut new interfaces through the primary module structure", and that a tool can compute these interfaces once a system has been assembled [24]. This means that a module is no longer sovereign over its own interfaces — rather, they are forced upon it by system composition. It follows immediately that modules cannot be changed independent of their use in a particular assembly, simply because it is unclear which interfaces to keep constant. This in turn hampers reuse in all cases in which a module is to be used in more than one composition. By contrast, what we have suggested here is much more conservative: we require that all interfaces of a module be made explicit at module design time, so that programmers can observe them while doing whatever they need to do, independently of each other.

8. Conclusion

It seems that implicit invocation with implicit announcement and modularity of components are in tension: one cannot be achieved without compromising the other. Inspired by how exception handling is done in Java, and how its interfaces-as-types provide for the decoupling of the caller from the called, we believe to have found a middle way that allows implicit announcement scoped to single classes, while achieving the classes' modularity through explicit interfaces. Applications of the so extended language are the same as that for other implicit invocation mechanisms with implicit or explicit event announcement, such as (database) triggers or occurrences of the event notification and observer patterns. Its limitations are clearly cases in which the publisher should remain unaware of the fact that it publishes. This includes, for practical reasons, some of the most prominent applications of AOP, in particular all extensively crosscutting concerns such as logging or tracing.

References

[1] J Aldrich "Open modules: modular reasoning about advice" in: *ECOOP* (2005) 144–168.

[2] P Avgustinov et al. “abc: an extensible AspectJ compiler” *TAOSD* (2005) 293–334.

[3] P Avgustinov et al. “Optimising AspectJ” in: *PLDI* (2005) 117–128.

[4] E Bodden, F Forster, F Steimann “Avoiding infinite recursion with stratified aspects” *NODe* (2006) to appear.

[5] C Clifton, GT Leavens “Observers and assistants: A proposal for modular aspect-oriented reasoning” in: *FOAL* (2002) 33–44.

[6] C Clifton, GT Leavens “Obliviousness, modular reasoning, and the behavioral subtyping analogy” in: *SPLAT* (2003).

[7] DS Dantas, D Walker “Harmless advice” in: *POPL* (2006) 383–396.

[8] J Dingel, D Garlan, S Jha, D Notkin “Reasoning about implicit invocation” in: *SIGSOFT '98/FSE-6* (1998) 209–221.

[9] T Elrad, RE Filman, A Bader “Aspect-oriented programming: Introduction” *CACM* 44:10 (2001) 29–32.

[10] E Ernst, DH Lorenz “Aspects and polymorphism in AspectJ” in: *AOSD* (2003) 150–157.

[11] KP Eswaran *Specifications, Implementations and Interactions of a Trigger Subsystem in an Integrated Database System* (IBM Research Report, RJ1820, 1976).

[12] PT Eugster, P Felber, R Guerraoui, AM Kermarrec “The many faces of publish/subscribe” *ACM Comput. Surv.* 35:2 (2003) 114–131.

[13] P Eugster "Type-based publish/subscribe: Concepts and experiences" *ACM Trans. Program. Lang. Syst.* 29:1 (2007).

[14] RE Filman, DP Friedman “Aspect-oriented programming is quantification and obliviousness” in: RE Filman et al. (eds) *Aspect-Oriented Software Development* (Addison-Wesley 2004).

[15] F Forster, F Steimann “AOP and the antinomy of the liar” in: *FOAL @ AOSD* (2006) 47–56.

[16] E Gamma, R Helm, R Johnson, J Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley 1995).

[] D Garlan, D Notkin “Formalizing design spaces: Implicit invocation mechanisms” in: *VDM '91: Formal Software Development Methods* Springer LNCS 551 (1991) 31–44.

[17] D Garlan, C Scott “Adding implicit invocation to traditional programming languages” in: *ICSE* (1993) 447–455.

[18] D Garlan, M Shaw *An Introduction to Software Architecture* CMU Software Engineering Institute Technical Report CMU/SEI-94-TR-21 (1994).

[19] W Griswold et al. “Modular software design with crosscutting interfaces” *IEEE Software* 23:1 (2006) 51–60.

[20] S Gudmundson, G Kiczales “Addressing practical software development issues in AspectJ with a pointcut interface” in: *Workshop on Advanced Separation of Concerns at ECOOP* (2001).

[21] E Hilsdale, J Hugunin “Advice weaving in AspectJ” in: *AOSD* (2004) 26–35.

[22] IEEE *Standard Computer Dictionary* (IEEE, 1991).

[23] G Kiczales, J Lamping, A Mendhekar, C Maeda, CV Lopes, JM Loingtier, J Irwin "Aspect-oriented programming" in: ECOOP (1997) 220-242.

[24] G Kiczales, M Mezini “Aspect-oriented programming and modular reasoning” in: *ICSE* (2005) 49–58.

[25] R Laddad AspectJ in Action (Manning 2003).

[26] J Ligatti, D Walker, S Zdancewic “A type-theoretic interpretation of pointcuts and advice” *Science of Computer Programming* 63:3 (2006) 240–266.

[27] B Liskov, JM Wing “A behavioral notion of subtyping” *ACM Trans. Program. Lang. Syst.* 16:6 (1994) 1811–1841.

[28] CV Lopes, P Dourish, DH Lorenz, K Lieberherr “Beyond AOP: Toward naturalistic programming”. in: *OOPSLA* (2003) 198–207.

[29] B Meyer Object-Oriented Software Construction 2nd Edition (Prentice-Hall 1997).

[30] L Mikhajlov, E Sekerinski “A study of the fragile base class problem” in: *ECOOP* (1998) 355–382.

[31] *MySQL 5.0 Reference Manual* (http://dev.mysql.com/).

[32] D Notkin, D Garlan, WG Griswold, KJ Sullivan “Adding implicit invocation to languages: Three approaches” in: *ISOTAS* (1993) 489–510.

[33] N Ongkingco et al. “Adding Open Modules to AspectJ” in: *AOSD* (2006) 39–50.

[34] H Ossher, P Tarr "Hyper/J: Multi-dimensional separation of concerns for Java" in: ICSE (2001) 729–730.

[35] K Ostermann, M Mezini, C Bockisch “Expressive point-cuts for increased modularity” in: *ECOOP* (2005) 214–240.

[36] H Rajan, KJ Sullivan “Eos: Instance-level aspects for integrated system design” in: *ESEC/SIGSOFT FSE* (2003) 291–306.

[37] H Rajan, KJ Sullivan “Classpects: Unifying aspect- and object-oriented language design” in: *ICSE* (2005) 59–68.

[38] H Rajan et al. “Preserving separation of concerns through compilation” in: *SPLAT Workshop @ AOSD* (2006).

[39] D Riehle “The Event Notification Pattern — Integrating implicit invocation with object-orientation” *Theory and Practice of Object Systems* 2:1 (1996) Page 43–52.

[40] F Steimann, P Mayer “Patterns of interface-based programming” *Journal of Object Technology* 4:5 (2005) 75–94.

[41] F Steimann “The paradoxical success of aspect-oriented programming” in: *OOPSLA* (2006) 481–497.

[42] M Störzer, J Graf “Using pointcut delta analysis to support evolution of aspect-oriented software” in: *ICSM* (2005) 653–656.

[43] KJ Sullivan et al. “Information hiding interfaces for aspect-oriented design” in: *ESEC/FSE* (2005) 166–175.

[44] C Szyperski *Component Software* (Addison-Wesley 1999).

[45] J Xu, H Rajan, KJ Sullivan “Understanding aspects via implicit invocation” in: *ASE* (2004) 332–335.

Part 2
Implementation of a compiler supporting Implicit Invocation with Implicit Announcement

9. Implementing a IIIA compiler

In the following the implementation of the compiler that extends Java with IIIA is documented.. This documentation contains the technical appendix details of Part I. Initially, in 9.1 some basic information about the AspectBench compiler [abc] frameworks are provided, because this compiler framework is used for the implementation. Afterwards, the concept of the implementation is introduced in 9.2. Finally, referring to the concept the extension of the AspectBench compiler is described in 9.3.

9.1. AspectBench compiler framework

The AspectBench compiler is a compiler framework which includes an implementation of the AspectJ language. The framework allows to extend and to optimize the basic AspectJ language via extensions [abc2005]. The compiler is based on Polyglot [Polyglot] and Soot [Soot], whereas Polyglot act as the front end of the compiler and Soot represents the back end. The front end parses the source code into an abstract syntax tree (AST) and is responsible for lexical, syntactical and semantic checks. These checks are performed in compiler passes which can also rewrite the AST. On the other hand the back end optimizes the AST and generates the executable code.

Polyglot

Polyglot is a compiler front end for Java. It is designed to be extend the Java language and to explore new language constructs for research. Therefore it allows to customize the grammar and the semantic analysis of a language. For customizing the syntax Polyglot includes a parser generator named PPG [PPG]. PPG is based on the CUP Parser generator for Java and provides the ability to extend an existing base language grammar. [Polyglot]

Soot

Soot can be seen as framework for manipulating, optimizing and transforming of Java byte code. The frameworks offers different representations of the Java byte code to perform different tasks, namely manipulation, optimization, decompilation and inspection of byte code. [Soot] The AspectBench compiler uses Soot also for the process of weaving.

In summary it can be said that the AspectBench compiler combines Polyglot and Soot to offer a workbench for implementing AspectJ or AspectJ-based extensions. Therefore the AspectBench compiler offers the possibility to extend or adapt syntax, type system, semantic checks, source code transforms and byte code optimization of the AspectJ language. In order to get more information of AspectBench compiler extensions [abc2005] should be referred.

9.2. Outline of the implementation

As the IIIA extension of Java is based on language constructs introduced by AspectJ, it is reasonable to re-use the existing functionality of the AspectJ compiler. For this purpose it is necessary to define how the IIIA language constructs map to Java or AspectJ language constructs and how the transformation of a IIIA program into an AspectJ program is accomplished, including the semantic checks that are needed. This general proceeding of extending the AspectBench compiler is shown in section 3.4 of [abc2005]:

> *"The normal use of Polyglot is as a source-to-source compiler for extensions to Java, where the final rewriting passes transform new features into an equivalent pure Java AST. abc is different in that most of the transformation happens at a later stage, when weaving into Jimple. It is, however, often useful to employ Polyglot's original paradigm when implementing extensions to AspectJ that have an obvious counterpart in AspectJ itself."*

9.2.1. Basic mapping of IIIA to Java/AspectJ constructs

To transform an IIIA program into an AspectJ program it has to be defined how the constructs of the IIIA language map those of AspectJ and Java. An overview of the mapping can be found in Table 1.

IIIA construct	Java/AspectJ construct
exhibiting class	class + list of join point types + class local pointcuts definition
class local pointcut	pointcut - formal parameters + join point type name
join point type	class - methods - constructors
advising aspect	aspect + list of join point types
join point advice	advice (with exactly 1 join point type as formal parameter) - pointcut
"+" = additional elements , "-" = dropped elements	

Table 1: Mapping of IIIA to AspectJ/Java

Exhibiting classes are ordinary Java classes including a list of join point types (whose instances) the class exhibits. The class local pointcuts, defined within the exhibiting classes, are basically normal pointcut definitions known from AspectJ. However, they are defined without any formal parameters. Providing formal parameters would be redundant, as they are already determined through the field declarations of the join point type. For this reason a pointcut in IIIA needs to be linked to the join point type for which it is defined. This is done by adding the name of the join point type to

the pointcut definition. A join point type can be mapped to a normal class definition with the restriction, that the definition may only contain field declarations. Furthermore, final is the only modifier allowed for the fields.

Aspects map to normal AspectJ aspects, but analogous to classes they contain a list of join point types which the aspect may advise. An aspect may contain several advices. These advice definitions are AspectJ advice definitions with exactly one formal parameter typed with the corresponding join point type and without a pointcut. The pointcut for the advice is defined by the class local pointcut definitions which are scattered among the classes exhibiting the same join point type.

9.2.2. Basic transformation

In IIIA there is no central definition of a pointcut. The pointcut definition on which an advice relies is divided and every part is moved into the class definition the part belongs to. So every class exhibiting the corresponding join point type contains one part of the global pointcut definition for this join point type. The bracket holding together the scattered pointcut definitions is the join point type, because pointcuts on the class side as well as advices on the aspect side are each defined for one particular join point type.

To let the advice apply to every pointcut defined for the same join point type as the advice, the compiler collects all pointcut declarations for this join point type and combines them into one central pointcut definition. Serving as the interface between aspects and classes the proper location for the global pointcut definition is the join point type definition itself.

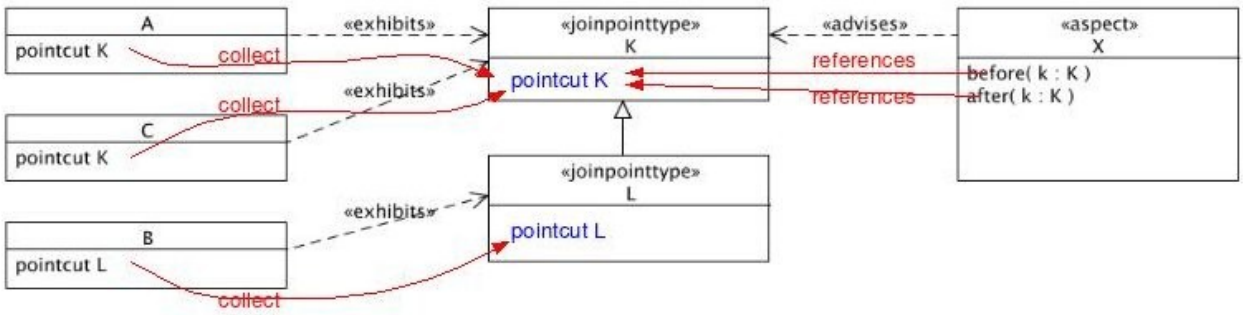

Figure 1: Simplified overview of transformation

The basic transforms is shown in Figure 1 and can be outlined as followed:

1. Collect all join point definitions and relate them to all classes exhibiting this join point type. To ask for information about a join point type, store the join point type definitions in a central repository which can be accessed by other compiler passes.

2. Collect all pointcuts for one particular join point type from all classes exhibiting this join point type and conjoin them into one global pointcut definition located within the join point type itself.

3. For each advice in each aspect set the pointcut of the advice to the global pointcut definition of the advised join point type as generated in step 2.

9.3. Extending the AspectBench Compiler

The extension of the AspectBench compiler is performed analogous to the procedure suggested in [abc2005]. This procedure can be briefly outlined as followed:

1. define the new syntax of the compiler extension
2. integrate new AST nodes and types
3. add new compiler passes, which modify the AST

Beginning with an overview of packages of the compiler extension, the following subsections will describe the implementation based on these three steps.

9.3.1. Package overview

In order to give a rough overview of the compiler extension the relevant packages are described. Figure 2 shows a overview of the packages.

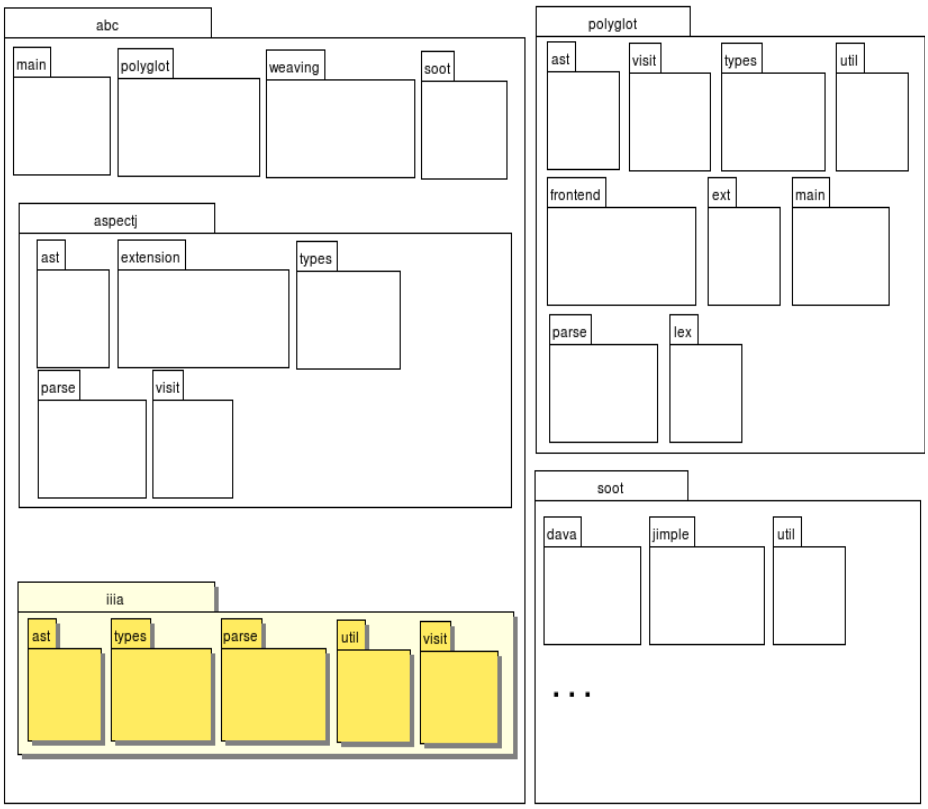

Figure 2: Package overview

The compiler extension is rooted in the package `abc.iiia`. This package contains a package for new or extended AST nodes (`abc.iiia.ast`), a package for new or extended types (`abc.iiia.types`) and a package for new compiler passes (`abc.iiia.visit`). Classes and resources for parsing an IIIA program are located in the package `abc.iiia.parse` and the `abc.iiia.util` package contains helper and utility classes. Furthermore the root package contains two classes which integrates the extension in the AspectBench compiler. The first class is called `AbcExtension` and the second one is called `ExtensionInfo`. Figure 3 shows the relationships of the two classes.

AbcExtension

This class extends the class `abc.main.AbcExtension`. This class configures and extends the behavior of the compiler. The two important extensions introduced by the `abc.iiia.AbcExtension` are:

1. create an `ExtensionInfo` object (see below)
2. initialize the lexer with new IIIA-keywords (see section 9.3.2)

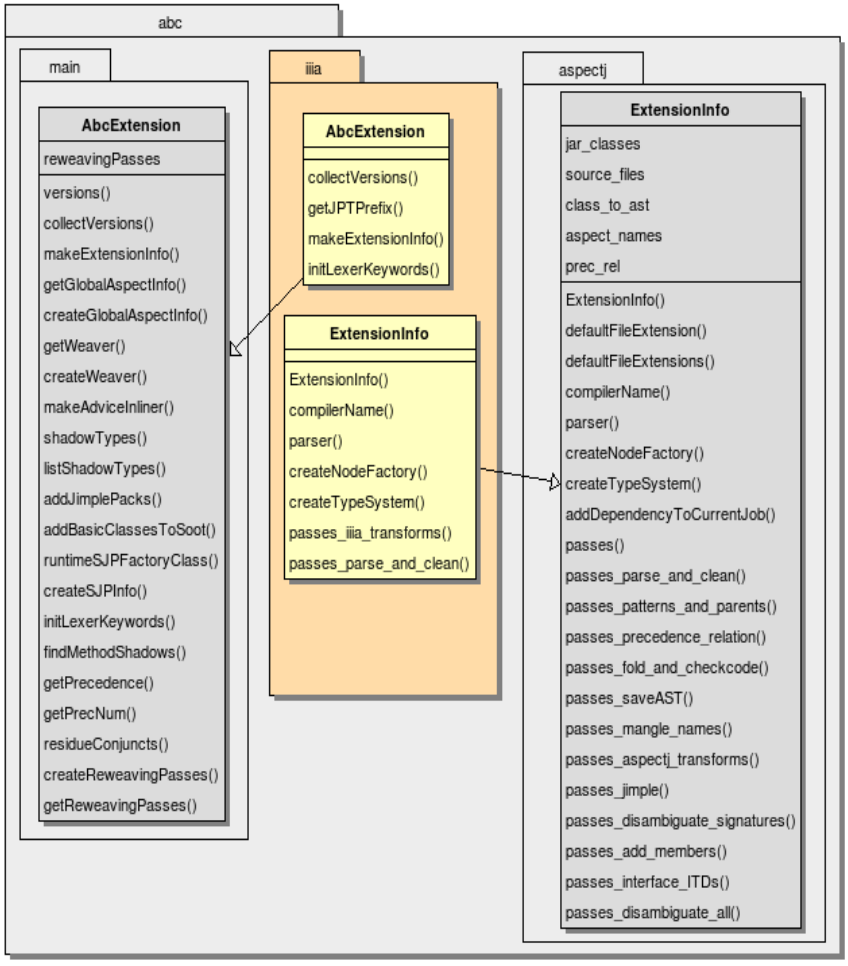

Figure 3: Extending and adapting the compiler behavior

ExtensionInfo

The `ExtensionInfo` is responsible for creating the `TypeSystem` and the `NodeFactory` (see 9.3.3). In order to offer new or adapted nodes and types the `ExtensionInfo` can provide specialized instances of the `NodeFactory` or `TypeSystem`. Also the `ExtensionInfo` defines which compiler passes traverses the AST and in which order the compiler passes traverses the AST. The integration of the new IIIA passes is described in 9.3.4.

9.3.2. Extending parser and lexer

The front end of the compiler is responsible for the parsing of the source code and the generation of the AST. The AspectBench compiler uses Polyglot as front end for parsing, which includes a parser generator named PPG. PPG is based on the CUP parser generator for Java and allows to extend a base language grammar [Polyglot].

The contribution of PPG is described in [PPG]:

> *"PPG is a parser generator for extensible grammars, based on the CUP parser generator. It provides the ability to extend an existing base language grammar written in CUP or PPG with localized, easily maintained changes."*

In order to use PPG for generating a new parser the new syntax has to be defined in PPG notation. The PPG notation is based on the CUP notation, but extends the CUP notation in some points to allow the extension of a base language grammar. The exact notation of PPG and CUP can be found at [PPG] and [CUP].

As IIIA is based on the AspectJ syntax it is reasonable to use the AspectJ syntax as base grammar and to modify it in such a way that the IIIA syntax is supported. Therefore the AspectJ syntax definition is included.

The second step for adapting the AspectJ syntax is the definition of new tokens[2] for the new keywords. The IIIA syntax introduces the following tokens:

```
terminal  Token JOINPOINTTYPE;
terminal  Token EXHIBITS;
terminal  Token EXHIBIT;
terminal  Token ADVISES;
```

Listing 1 : New terminal syntax token

The token `JOINPOINTTYPE` is used to define a new join point type. The token `EXHIBITS` offers classes the possibility to define which join point type they exhibit whereas aspects use `ADVISES` to define which join point types they advise. The token `EXHIBIT` is used for explicit exhibiting blocks.

After the definition of the new tokens also the new non-terminal tokens of the new syntax must be defined. A non-terminal token is a token that expands to other terminal or non-terminal tokens. Listing 2 shows the definition of the non-terminal tokens. As can be seen from the listing below a non-terminal token has to be typed. Thereby the types of a non-terminal tokens correspond to the classes of AST-nodes or the type `java.util.List`, when the token represents a list of nodes.

2 A token is the atomic part of the syntax

```
non terminal JoinpointDecl   joinpoint_declaration;
non terminal ClassBody       joinpoint_body;
non terminal List            joinpoint_field_declarations;
non terminal List            joinpoint_field_declarations_opt;
non terminal VarDeclarator   joinpoint_variable_declarator;
non terminal List            joinpoint_field_declaration;
non terminal Flags           joinpoint_modifier_opt;
non terminal Flags           joinpoint_modifier;
non terminal JoinpointName   joinpointtype_name;
non terminal List            joinpointtype_name_list;
non terminal List            exhibition;
non terminal List            exhibition_opt;
non terminal ExhibitBlock    exhibit_block;
non terminal List            advising;
non terminal List            advising_opt;
non terminal List            joinpointadvice_declaration_list;
non terminal List            polymorphic_pc_declaration;
```

Listing 2 : New non-terminal syntax tokens

Having all new terminal and non-terminal tokens integrated into the list of allowed tokens, the syntax of the base grammar can be adapted:

1. **The syntax is defined by combining (terminal and non-terminal) tokens**

   For the newly introduced non-terminal tokens the syntax can be defined by using CUP notation. To adopt the syntax of old tokens the PPG extension of the CUP notation must be used. With this extension it can be specified how the inherited grammar for this token should be handled. In order to preserve the parsers ability to parse “normal” AspectJ, only the *extend* specification is used. This specification adds the newly defined grammar to the original grammar, but does not change it.

2. **Embedded Java code creates the AST node for this token.**

   The AspectBench compiler uses a node factory[3], which is responsible for the creation of the AST nodes. In the embedded Java code grabs the tokens of the syntax, and uses the node factory to create from these tokens a new AST node instance.

To illustrate the described procedure the extension of the class declaration syntax is shown exemplary in Listing 3. The normal class declaration is extended and the extension adds the non-terminal token `exhibition` to the declaration.

```
extend class_declaration ::=
          // ClassDecl
        modifiers_opt:a
        CLASS:n IDENTIFIER:b
        super_opt:c
        interfaces_opt:d
        exhibition:f
      class_body:e
   {:
      Grm.parserTrace("CLASS declaration "+ b +" exhibits "+f+" advises ");
      RESULT = parser.nf.ClassDecl(parser.pos(n, e),a, b.getIdentifier(), c, d, f, new
ArrayList() , e);
   :}
;
```

Listing 3 : Class declaration in PPG syntax

3 The node factor is described in more detail in the next section.

The definition of the token `exhibition` is listed as an example for the grammar definition of a new (non-terminal) token. The definition in Listing 4 consists of the terminal token `EXHIBITS` and the non-terminal token `joinpointtype_name_list`.

```
exhibition ::=
        // List of JoinpointName
     EXHIBITS:e joinpointtype_name_list:l
     {: RESULT =  l; :}
   ;
```

Listing 4 : Class exhibition in PPG syntax

A `joinpointtype_name_list` is simply a comma-separated list of `joinpointtype_name` tokens (Listing 5).

```
joinpointtype_name_list ::=
     // List of join pointName
   join point_name:a
     {:
       List l = new TypedList(new LinkedList(), join pointName.class, false);
       l.add(a);
       RESULT = l;
     :}
    |   join point_name_list:a COMMA join point_name:b
     {:
       RESULT = a;
       a.add(b);
     :}
   ;
```

Listing 5 : Join point type list definition in PPG syntax

This example describes the general procedure of the syntax definition. The full syntax definition in PPG notation can be taken from the file `iiia.ppg` within the package `abc.iiia.parse`. A complete definition of the syntax in BNF can be found in Listing 6.

```
type_specifier ::=
   ...
   | joinpointtype_name

joinpointtype_name ::=
   identifier | package_name "." identifier

type_declaration ::=
   ...
   | joinpoint_declaration

joinpoint_declaration ::=
   "joinpoint type" identifier
   "{" { join point_field_declaration } "}"

joinpoint_field_declaration ::=
   [ "final" ] type identifier ";"

class_declaration ::=
   { modifier } "class" identifier
   ...
   [ "exhibits" joinpointtype_name [ { "," join point_name } ] ]
   "{" { class_body } "}"
```

```
class_body ::=  { class_member }

class_member ::=
   ...
   | joinpoint_pointcut_declaration

block ::=
   ...
   | exhibit_block

exhibit_block ::=
   "exhibit new " joinpointtype_name "(" [ {argument } ] ")"
   "{" statement "}"

joinpoint_pointcut_declaration ::=
   "pointcut" joinpointtype_name ":" pointcut_expression ";"

pointcut_expression ::=
   ...
   | "super"

aspect_declaration ::=
   { modifier } "aspect" aspect_name
   [ "advises" joinpointtype_name [ { "," joinpointtype_name } ] ]
   "{" { joinpoint_advice_declaration } "}"

joinpoint_advice_declaration ::=
   ("before"|"around"|"after")
   "(" joinpointtype_name variable_declarator ")
   {" advice_content "}"
```

Listing 6 : Complete syntax definition of IIIA in BNF

Once the syntax is defined in PPG notation, the generation of the parser consists of two steps. Firstly the file including the syntax definition is fed into PPG (class: `ppg.PPG`) in order to create from the syntax definition and the included base grammar a syntax definition in CUP notation. Secondly the generated CUP syntax definition is fed into CUP itself (class: `java_cup.Main`) in order to generate the parser. The CUP parser generator produces two Java source files representing the parser. The first Java file is a table of symbols where every terminal token is mapped to an integer. The second file represents the parser rules. These rules uses the symbols and the Java code which was embedded in the token definitions to generate the AST nodes.

```
package abc.iiia;

public class AbcExtension extends abc.main.AbcExtension {
   ...
   public abc.aspectj.ExtensionInfo makeExtensionInfo(Collection jar_classes,Collection
aspect_sources) {
      return new abc.iiia.ExtensionInfo(jar_classes, aspect_sources);
   }

   public void initLexerKeywords(AbcLexer lexer) {
      super.initLexerKeywords(lexer);

      lexer.addGlobalKeyword("joinpointtype", new LexerAction_c(new Integer(sym.join
point)));
      lexer.addJavaKeyword("exhibits", new LexerAction_c(new Integer(sym.EXHIBITS)));
      lexer.addJavaKeyword("exhibit", new LexerAction_c(new Integer(sym.EXHIBIT)));
      lexer.addAspectJKeyword("advises", new LexerAction_c(new Integer(sym.ADVISES)));
   }
}
```

Listing 7 : Adaptation of lexer

The AspectBench compiler connects the PPG parser with a lexer in order to map the tokens from the syntax definition to keywords which can be used in the source code. This mapping is performed in the lexer of the AspectBench compiler. The lexer is initialized within the class `abc.main.AbcExtension`. As every compiler extension has to subclass this class every extension is able to extend or adapt the lexer's initialization. This mapping is performed in the method `initLexer(AbcLexer)`.The mapping of IIIA symbols to keywords can be seen in Listing 7.

9.3.3. Introducing new AST nodes and extending the type system

By extending Java with IIIA there is the need for new AST nodes representing the adapted language elements. As IIIA is an AspectJ based extension for Java the set of new nodes extends the set of existing nodes from the Java or AspectJ language. An AST node in the AspectBench compiler consists of an interface describing the services of an node and an implementing class. The decoupling of the interface of an node from its implementation is necessary to allow a node to be subtype of multiple other nodes.[4] Figure 4 shows the typical relationships of a node.

Figure 4: Design of AST nodes

The nodes are instantiated by a node factory. This factory should be the only component of the compiler knowing the nodes' implementations. All other components like the parser or compiler passes only know the interfaces of the nodes. Therefore the node factory must offer a factory method for every node which should be instantiated. The AspectBench compiler allows every compiler extension to offer a specialized node factory. With this node factory new nodes can be introduced or creation of existing nodes can be changed.

4 This is done to let the node's interface extend multiple node interfaces.

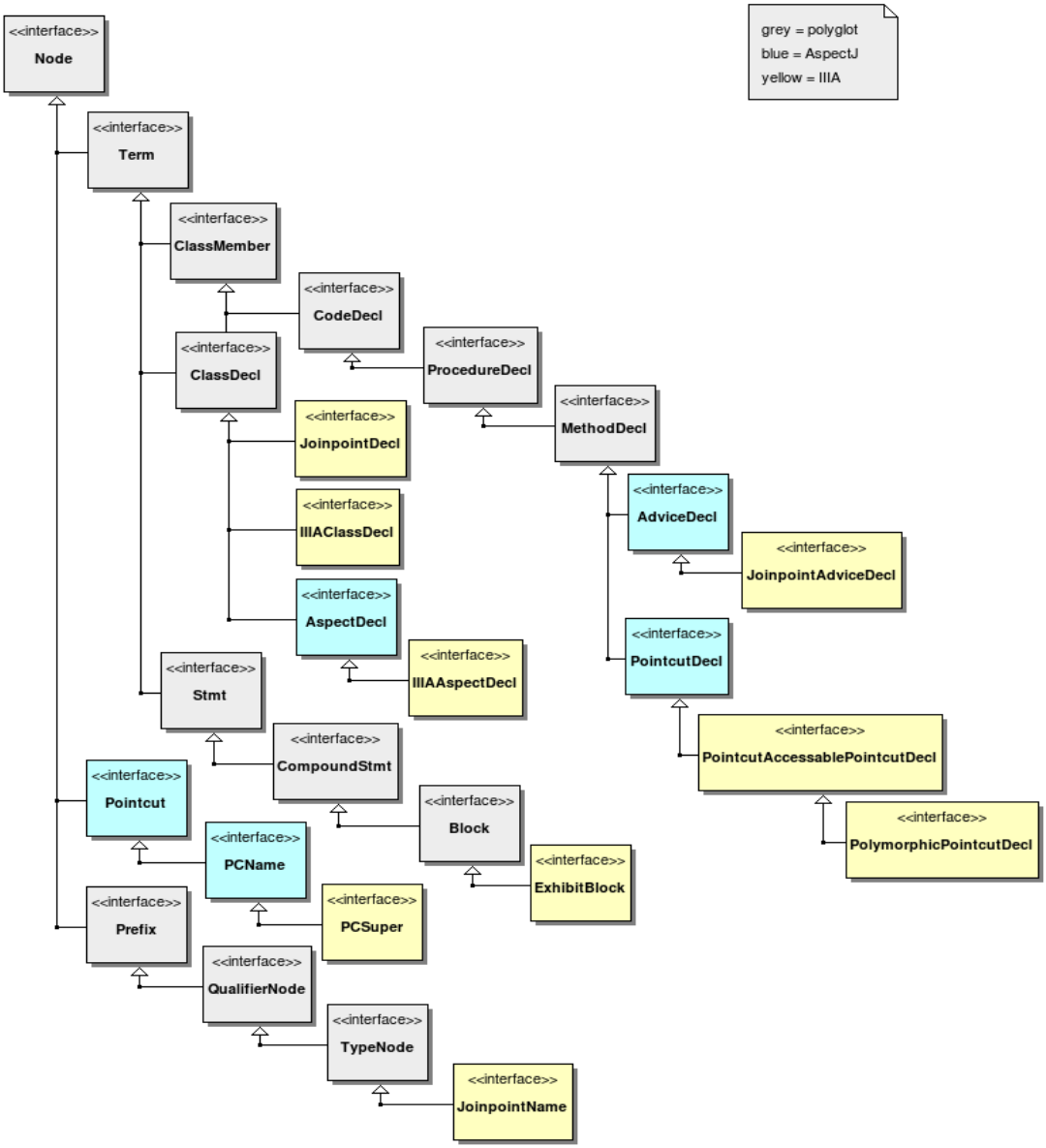

Figure 5: Hierarchy of interfaces of IIIA-AST nodes

Figure 5 shows the hierarchy of the nodes introduced by IIIA. In the following these nodes are described in more detail.

JoinpointDecl

As a join point type maps to a normal Java class its AST node extends the `ClassDecl` node. The inherited behavior of the node is unchanged. However, nodes of this type are linked to the `IIIAClassDecl` nodes of all classes exhibiting this join point type. These links are used for generating the join point type's global pointcut definition from all class local pointcuts[5]. Figure 6 shows the `JoinpointDecl` node in a class diagram.

5 This will be described in 9.3.4 in more detail.

JoinpointName

A new `TypeNode` is required for referencing a join point type (e.g. in a class node which lists all exhibited join point types, see above). This type node's interface is called `JoinpointName` and is implemented by two classes, the `AmbJoinpointName_c` and the `CanonicalJoinpointName_c`, whereas the former is used if the type of the node is ambiguous and the latter, if the the node's type is unambiguous (Figure 6). Unambiguous types contain the fully qualified name of the type. For example `String` is a ambiguous representation of a type, whereas `java.lang.String` is its unambiguous representation.

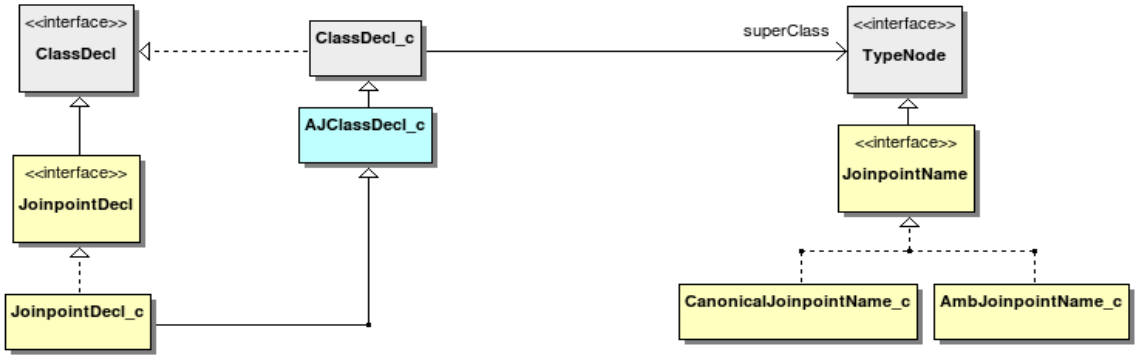

Figure 6: AST nodes - JoinpointDecl, JoinpointName

IIIAClassDecl

As classes are able to exhibit multiple join point types the class nodes have to be able to store a list of names of join point types. Therefore it is necessary to extend the `ClassDecl` node. In Figure 7 this extension is illustrated.

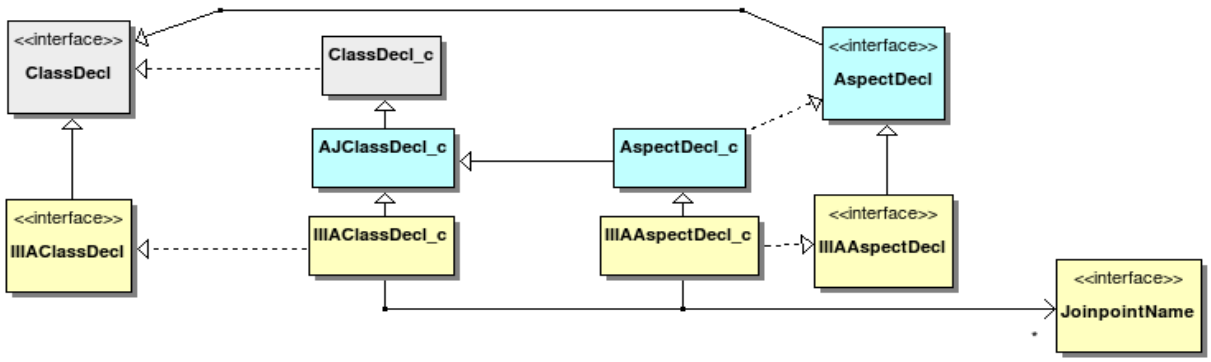

Figure 7: AST Nodes - IIIAClassDecl, IIIAAspectDecl

IIIAAspectDecl

Analogous to the class declaration node, the aspect declaration node must also be extended to enable the storing of a list of join point type names. This list defines which join point types the aspect advises. Figure 7 shows extended node.

ExhibitBlock

For explicitly announced join points, an extended block node is needed to store the block's link to a join point type. This link is necessary to determine the join point type fields and to generate the pointcut matching this block (see Figure 8).

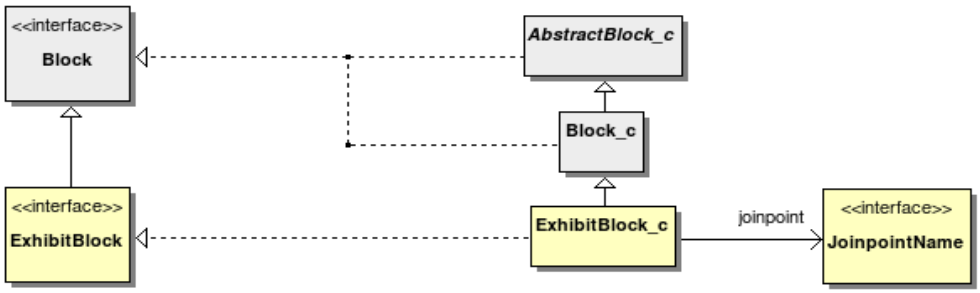

Figure 8: AST nodes - ExhibitBlock

ArgsExtractablePC

In AspectJ variables of the context of a join point are exposed by pointcut definitions. In IIIA pointcuts are linked to a join point type, which can contain one or more fields. Therefore it is necessary that each pointcut exposes a variable for each field of the join point type it is associated with. So, this interfaceThis is checked by the interface `ArgsExtractablePC`, which defines a method providing this service. The interface has to be implemented by all context-exposing pointcut nodes. These are `PCArg` (in AspectJ: `arg()`), `PCBinary` (in AspectJ conjunction or disjunction of pointcuts via `&&` or `||`), `PCTarget` (in AspectJ: `target()`) and `PCThis` (in ApsectJ `this()`). These pointcut nodes have to be extended, because the existing nodes classes do not implement this interface and therewith do not provide its service. Figure 9 shows the context of this interface in a class diagram.

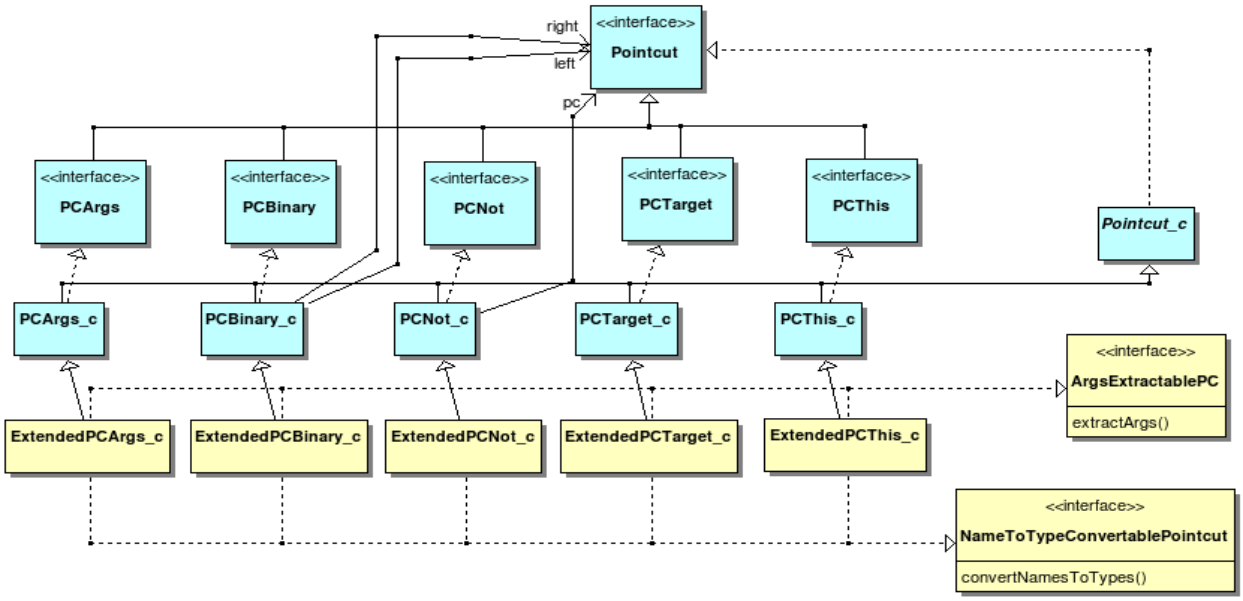

Figure 9: AST nodes - Pointcut nodes

NameToTypeConvertablePointcut

Pointcuts of a join point type are not allowed to match join points matched by a pointcut of a join point subtype defined in the same class. So the pointcut of the join point supertype must be restricted. This restriction can be achieved by negate the pointcut of the join point subtype and join the pointcut of the join point supertype with this negated pointcut. But the join point subtype may contain additional fields so that the pointcut of the join point subtype exposes additional variables. For this reason the pointcut of the join point subtype must be converted before it is negated and joined with the pointcut of the join point supertype. The conversion of the pointcut replaces all variable names within the pointcut with the types of the variables. The types are determined by the field declarations of the join point subtype. After this conversion the pointcut of the join point supertype can be restricted accordingly. The context of the interface is illustrated in Figure 9.

PCSuper

The PCSuper node is used when referencing the pointcut of the same join point type from the superclass with the keyword super. Actually this node is only a simple placeholder and marker for a special compiler pass. When the compiler pass traverses the AST and visits a super pointcut node, it replaces it with the pointcut definition from the super class. The role of the node is described in 9.3.4 in context with the `PolymorphicPointcutInheriter`. Figure 10 shows the node's relationships.

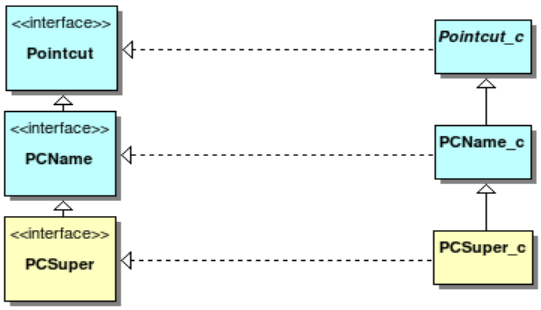

Figure 10: AST nodes - PCSuper

JoinpointAdviceDecl

The AdviceDecl node has to be extended to make an association of an advice to a join point type possible. This association is needed to determine the fields of the join point that are used as advice formals. Within the scope of subtyping of join point types the advices must be able to store a second association to a join point type. This is necessary, when the compiler pass `JoinpointSubtypeAdviceGenerator` copies an advice to simulate subtyping of join point types (see 9.3.4 for more information). The relationships of the `JoinpointAdviceDecl` node are shown in Figure 11.

PolymorphicPointcutDecl

Because a class local pointcuts are defined for a certain join point type, an extended pointcut node is required. This extended pointcut node can be associated with a join point type. This can be seen in Figure 11.

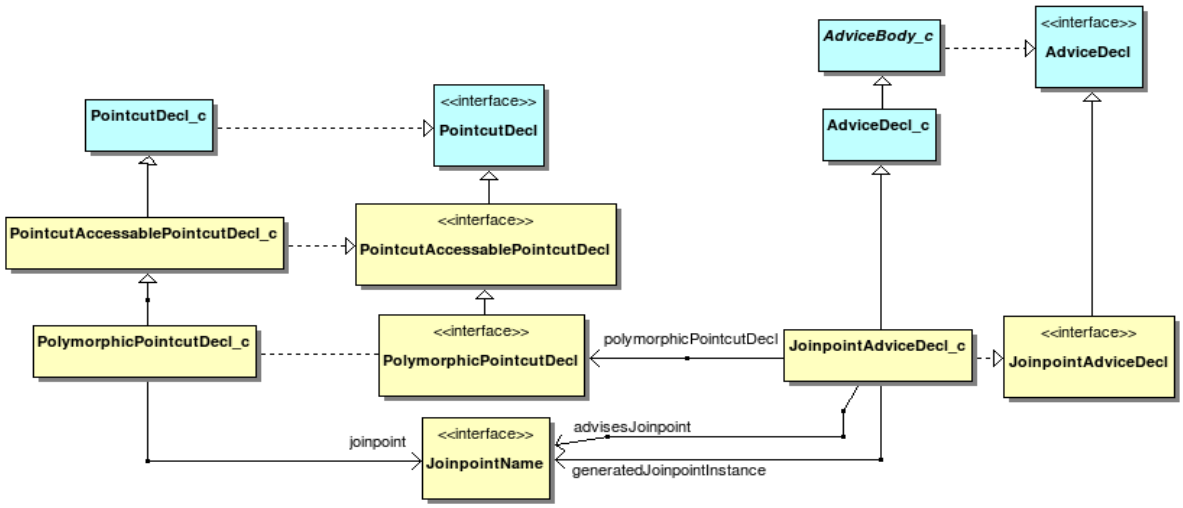

Figure 11: AST nodes - PolymorphicPointcutDecl, JoinpointAdviceDecl

For the implementation of the compiler which support the approach of IIIA also the type system of the AspectBench compiler must be adapted. The type system of the AspectBench compiler is described in [PolyglotDoc] as follows:

> *"A type system object acts as a factory for objects representing types and related constructs such as method signatures. The type system object also provides some type checking functionality."*

This type system object is also used in Polyglot's compiler pass `TypeBuilder` which builds type objects for the nodes of the AST. Normally the `TypeBuilder` creates for `ClassDecl` nodes a `ParsedClassType`. In order to simplify semantic checks the `ParsedClassType` is extended by `ParsedIIIAClassType`. This class type stores the list of join point types a class exhibits.

Furthermore the type system object is used build up a central repository of join point type definitions. These repository is implemented as `JoinpointtypeManager` and is filled by the IIIA compiler pass `JoinpointCollector` (see 9.3.4). This repository is used by several IIIA compiler passes to ask for information about join point types. Some of the passes need this information in order to perform their tasks.[6] Figure 12 shows the extension of the type system and the embedding of the `JoinpointtypeManager`.

6 This will be described in 9.3.4.

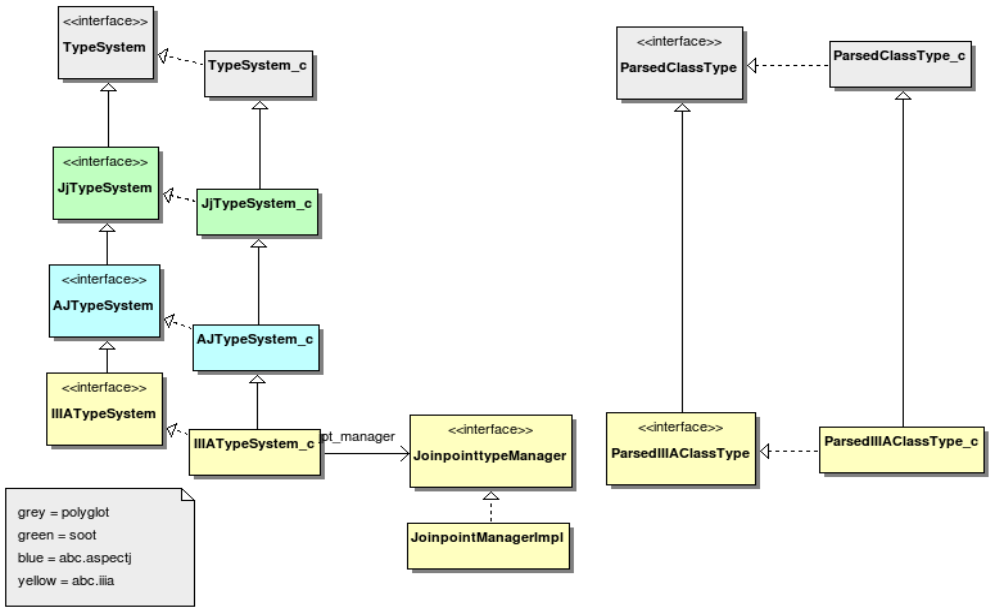

Figure 12: Extending the TypeSystem

9.3.4. Introducing new compiler passes

For the transformation of the AST Polygot uses compiler passes. The single passes traverse and rewrite the AST consecutively while each pass is using the output of the previous pass as input. Each AspectBench extension can create a list of compiler passes, which defines which passes traverse the AST and in which order they are executed. As there are multiple passes every pass performs only a small amount of work and it is easy to insert new passes or change the list of compiler passes. [abc2005].

The adaptation of the list of passes is done in the class `abc.iiia.ExtensionInfo`. In this class the new IIIA compiler passes are inserted into the list of the remaining AspectBench compiler passes. In order to be able to use the existing AspectBench compiler passes with as few changes as possible. Therefore, the IIIA passes are inserted directly after the parsing, which actually generates the AST, and the compiler pass `TypeBuilder`, which uses the type system for creating type objects for the AST nodes. After the IIIA passes, which transform the IIIA-AST into a pure AspectJ-AST, the original compiler passes of the AspectBench compiler are inserted, which produce from the pure AspectJ-AST executable byte code.

Figure 13 shows the new compiler passes in the order they are executed in a class diagram. To show the collaboration of the passes their, transformation is illustrated step by step with a simple example. The source code of the example can be found in Listing 8, whereas Figure 14 shows the (simplified) AST which the parser generates from the source code of Listing 8. To improve the readability of the object diagrams, nodes (or subtrees of nodes), which are not involved in the actual transformation, are labeled with "...". Modified, newly inserted or replaced nodes are highlighted in red.

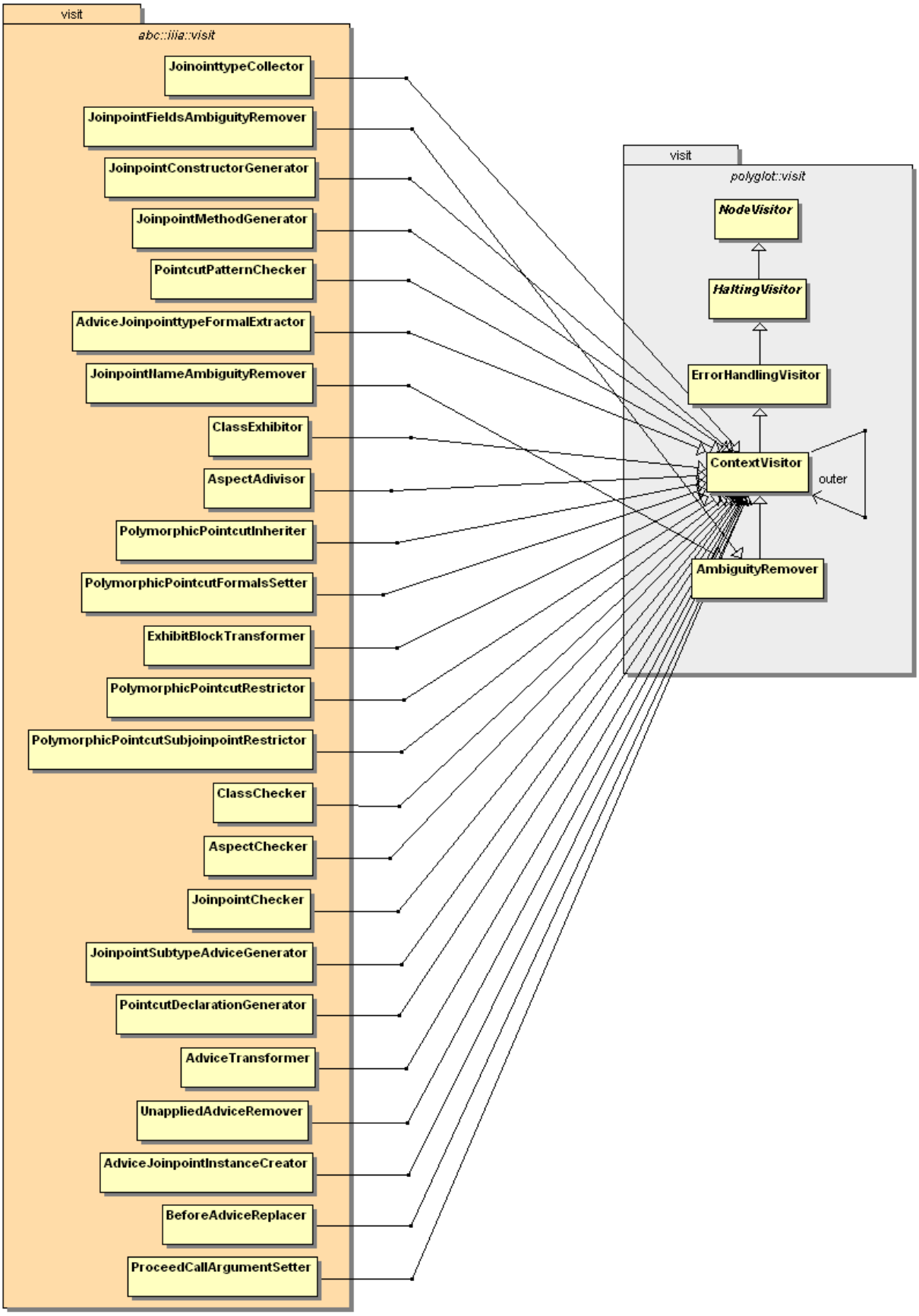

Figure 13: New compiler passes

```
joinpointtype J {}
joinpointtype K extends J {final int a}
joinpointtype L extends K {String b}

class A exhibits J,L{
  pointcut J : execution(void doA1());
  pointcut K : execution(void doA2(..)) && args(b,a);

  void doA1() {
     ...
  }
  void doA2(String name, int number) {
     System.out.println(name+" - "+number);
     ...
  }
  void doA3() {
     final int n = 12;
     final String m = "Hallo";
     ...
     exhibit new L (n,m) {
        System.out.println("...");
     }
     ...
  }
}

class B extends A exhibits J,K {
  pointcut J : super || ( execution(void doB1()) );
  pointcut K : execution(void doB2(..)) && args(..,a);

  /*@Override*/
  void doA1() {
     ...
  }

  void doB1() {
     ...
  }

  void doB2(String name, int i) {
     ...
  }
}
aspect X advises J,K {

  before(J j) {
     System.out.println("Before J");
  }

  void around(K k) {
     ...
     proceed();
     ...
  }
}
```

Listing 8 : Transformation example - source code

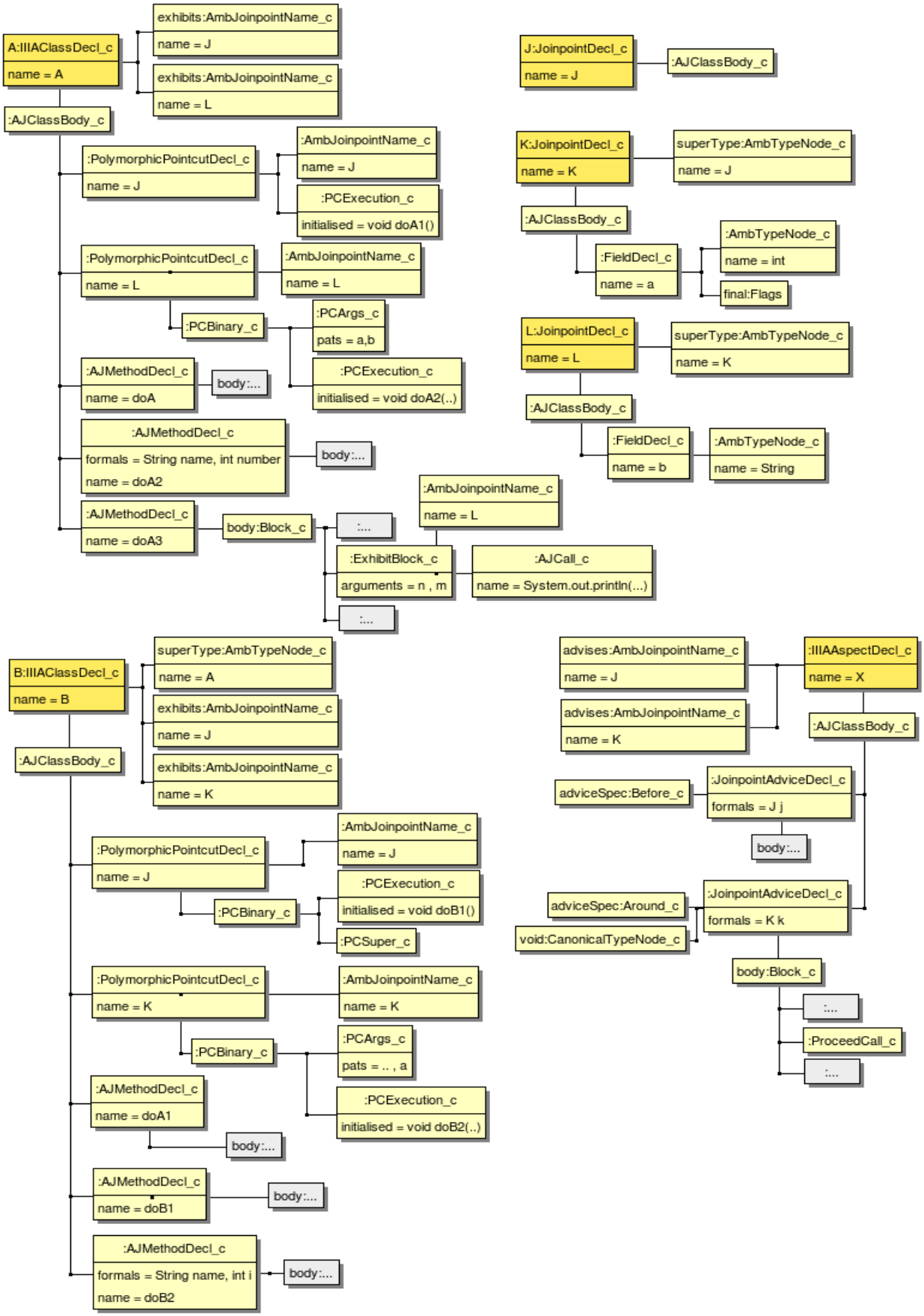

Figure 14: Transformation - Initial AST

Pass 1: JoinpointCollector

In the first pass all `JoinpointDecl` nodes are collected and stored in the `JoinpointManger` within the types system object. Therewith the type system offers access to a central repository for all join point types. This repository can be used by all compiler passes to retrieve information about a join point type. The `JoinpointtypeManager` stores a `JoinpointDecl` node by using its type as key in order to have a unique key for looking up a the corresponding `JoinpointDecl` node.

Pass 2: JoinpointFiledsAmbiguityRemover

The ambiguity of join point type fields is removed in order to make it possible to generate the constructors for the join point instances in the next pass. It is important to remove the ambiguity of the fields before the constructors are generated, because the constructor of a join point subtype contains also the fields of its join point supertype, although the types of the join point supertype fields may not have been imported in the source. This would result in a compiler error as the compiler can not resolve the types. To avoid these errors the ambiguity of the types is removed by replacing the simple names of the types with their fully qualified names.

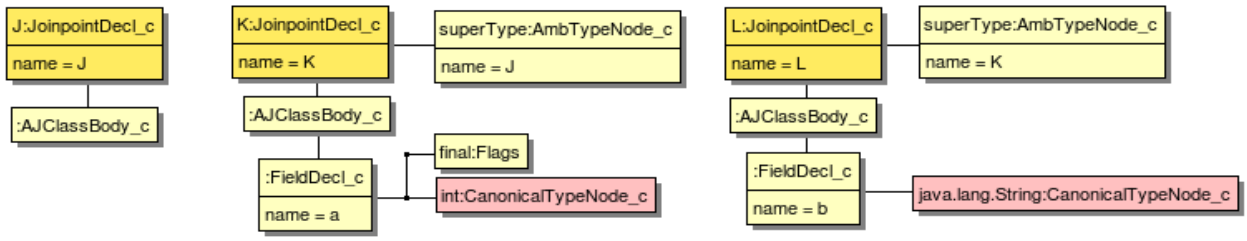

Figure 15: Transformation - JoinpointFieldAmbuigityRemover

Figure 15 shows the result of the transformation in the example. The type nodes of the `FieldDecl` nodes are replaced by `CanonicalType` nodes, which in the AspectBench compiler represents the fully qualified name of a type. This is noticeable at the change from `String` to `java.lang.String`.

Pass 3: JoinpointConstructorGenerator

For each `JoinpointDecl` node a constructor is generated. The generated constructor is used to create an instance of the join point type within advices. To ensure that all fields of the join point type are initialized the constructor expects for each (inherited and not inherited) field one formal parameter. The constructor of a join point subtype delegates the initialization of inherited fields to the constructor of its join point supertype and initializes its own fields afterwards. The actual instantiation of the join point type within the advices is done in the `AdvideJoinpointInstanceCreator`-pass.

Applied to the above example this transformation causes the generation of one constructor for each join point type: an `AJConstructorDecl` node is added to the body of the `JoinpointDecl` node. Figure 16 shows the result of the transformation. As it can be seen in the constructor of join point type `L`, it calls the constructor of `K` in order to initialize the field `a` and initialize field `b` by itself.

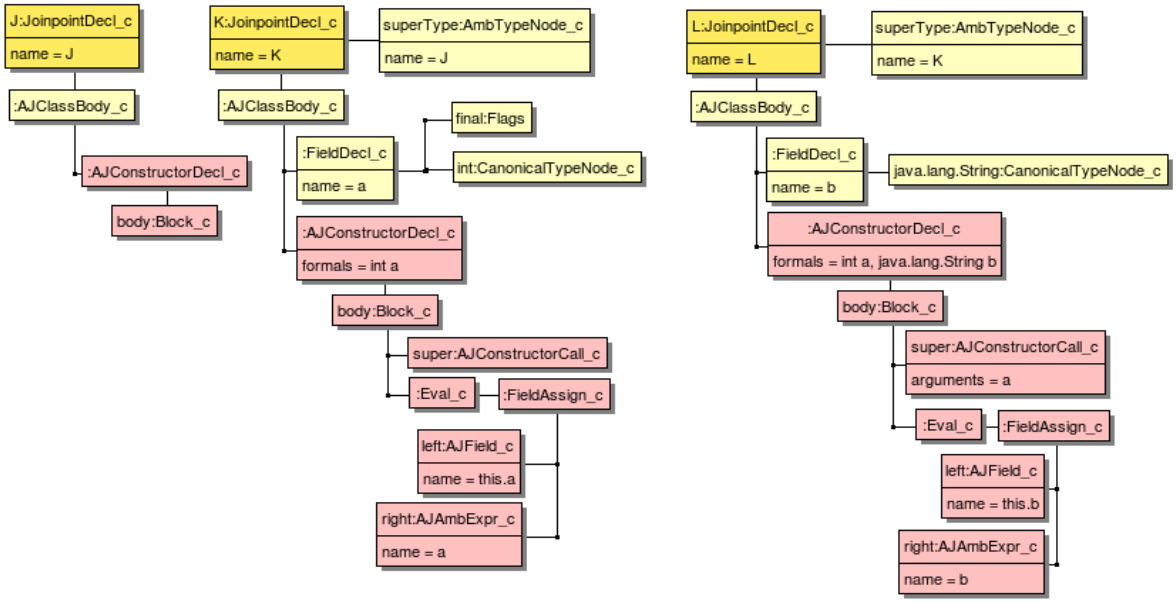

Figure 16: Transformation - JoinpointConstructorGenerator

Pass 4: JoinpointMethodGenerator

An exhibit block is transformed into an anonymous inner class of the join point type exhibited by the block. In order to create an anonymous inner class of the join point type a method within the join point type is needed which can be overridden with the code of the exhibit block. Therefore a method with an empty body is added to the body of the join point type. This method may not be `abstract`, because if the join point type contains a abstract method itself must be abstract, too. Being an abstract type the creation of a join point type instance via a constructor would not be possible without creating a concrete subtype of it. For this reason an empty method is generated. This method requires for each of the join point type's field (including those inherited) one formal parameter. This method will be overridden in the anonymous inner class that is created when the `ExhibitBlockTransformer` found an `ExhibitBlock` node while traversing the AST.

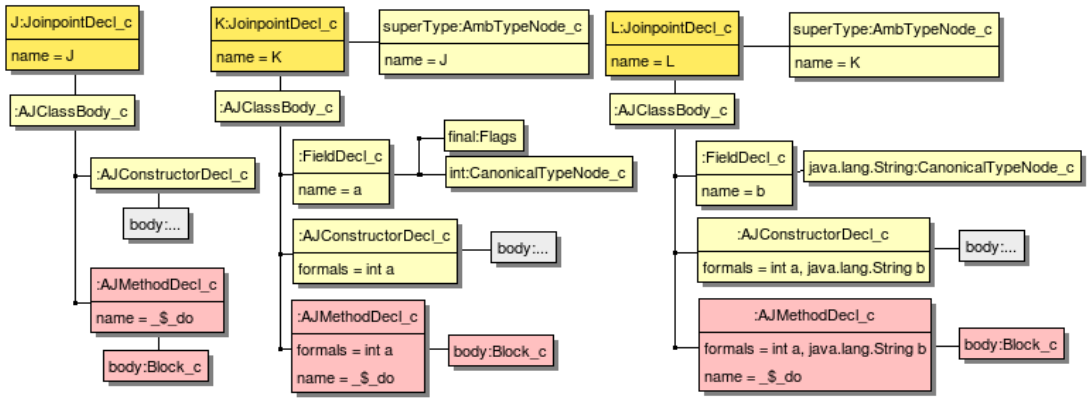

Figure 17: Transformation - JoinpointMethodGenerator

In the example this transformation generates an `AJMethodDecl` node named with `_$_do` for each join point type and adds the node to the body of the corresponding `JoinpointDecl` nodes. The generated methods contain an empty body and requires for each field a formal parameter. Figure 17 shows the result of this transformation.

Pass 5: PointcutPatternChecker

This pass simply checks the pointcut patterns if they adhere to the required syntax of IIIA . In particular, a class pattern in an `execution()`-pointcut within a `PolymorphicPointcutDecl` node is not allowed, because the hole pointcut is restricted to match only within the class by adding a corresponding `within()`-pointcut to the pointcut definition in the `PolymorphicPointcutRestrictor` pass.

Pass 6: AdviceJoinpointFormalExtractor

This pass examines each `JoinpointAdvice` node and checks if there is only one formal parameter defined for the advice, which is represented by this node, and if the formal parameter is a join point type. If this check succeeds, the name of the join point type is stored in the `JoinpointAdviceDecl` node, so that later passes know to which join point type the advice belongs. Also the name of the formal parameter is stored in the node to make the instance of the join point type available under this name. If the check fails, a compiler error is generated.

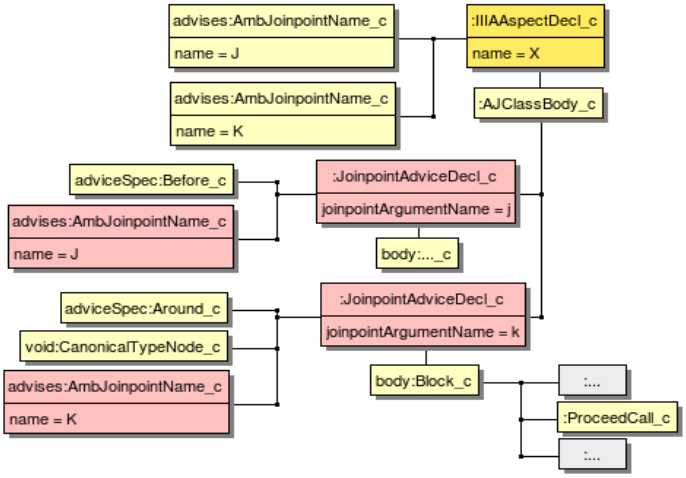

Figure 18: Transformation - AdviceJoinpointtypeFormalExtractor

The transformation of this pass is shown in Figure 18. The formal parameters of both advice are removed and an association to a corresponding `JoinpointName` node is added. Furthermore the names of the formal parameters are stored in the appropriate `JoinpointAdviceDecl` node.

Pass 7: JoinpointNameAmbiguityRemover

The ambiguity of the `JoinpointName` nodes is removed by replacing them with their fully qualified names. This is necessary for looking up the corresponding `JoinpointDecl` nodes in the `JoinpointtypeManager`. The simple name is not adequate to look up in the repository, because there can be several join point types defined with equal names but in different packages.

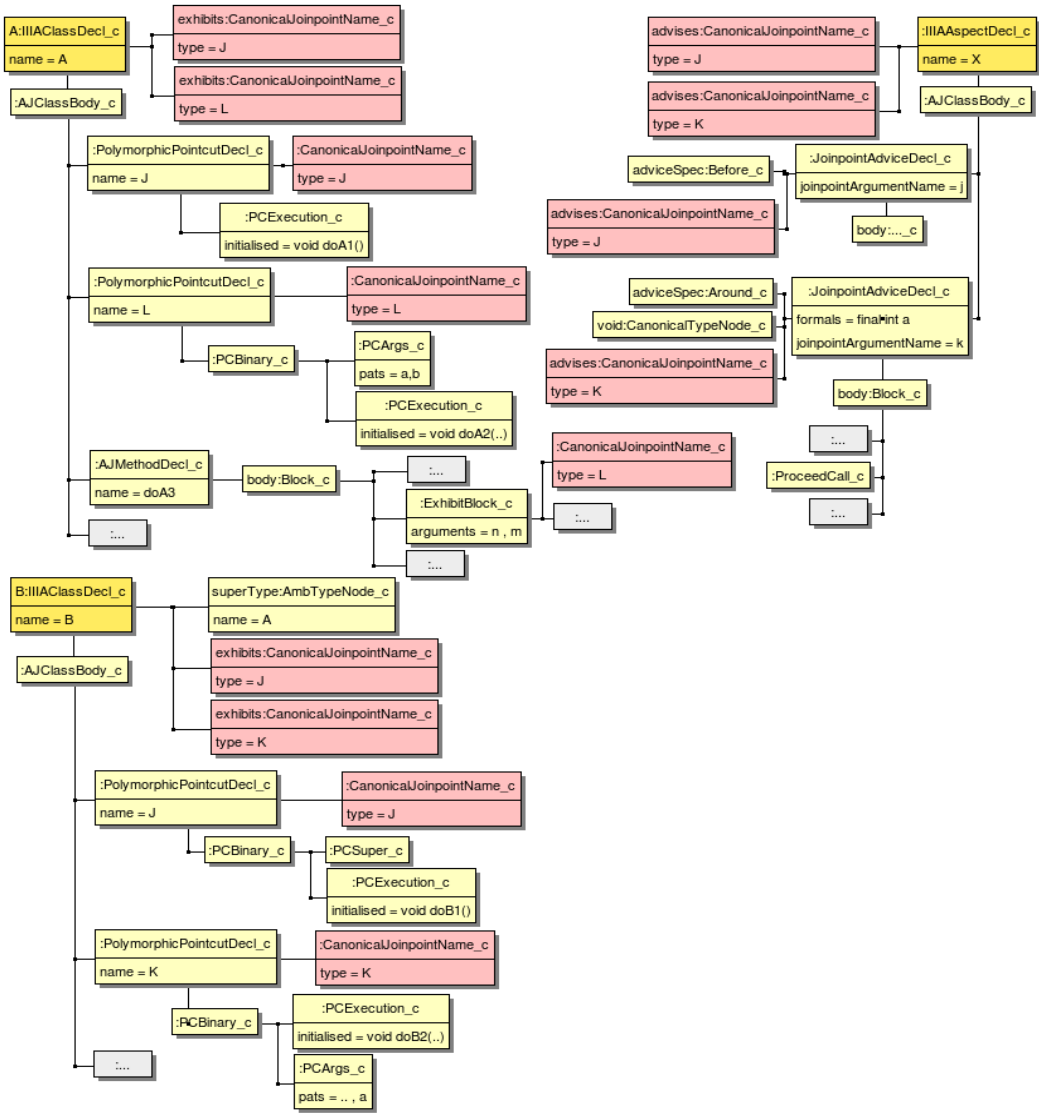

Figure 19: Transformation - JoinpointNameAmbiguityRemover

In the example AST all occurring `AmbJoinpointName` nodes are replaced with `CanonicalJoinpointName` nodes. These nodes represent the fully qualified names of the join point types. Figure 19 shows the transformed AST.

Pass 8: ClassExhibitor

The `ClassExhibitor` pass is responsible for two tasks: 1. the pass adds the list of `JoinpointName` nodes to the (extended) class type. This information is used during later semantic checks. 2. The `JoinpointDecl` nodes of the join point types the class exhibits are associated with the `IIIAClassDecl` nodes. These associations are required in later passes to collect all pointcuts of one join point type.

Pass 9: AspectAdvisor

Analogous to the class nodes the `JoinpointDecl` nodes are associated with the `IIIAAspectDecl` nodes for all aspects advising the join point type. These associations are also needed for semantical checks.

Pass 10: PolymorphicPointcutInheriter

This pass resolves the references to super pointcuts by replacing `PCSuper` node with the pointcut of the superclass defined for the same join point type. The superclass's pointcut must be copied rather than referenced, because every pointcut will be restricted by a later pass to the class in which the pointcut is declared in. In particular, the superclass's pointcut will be restricted to match only within the superclass. If a reference would be used the subclass pointcut would never match as the restriction would be referenced, too. By copying the pointcut reference super gets the right semantics.

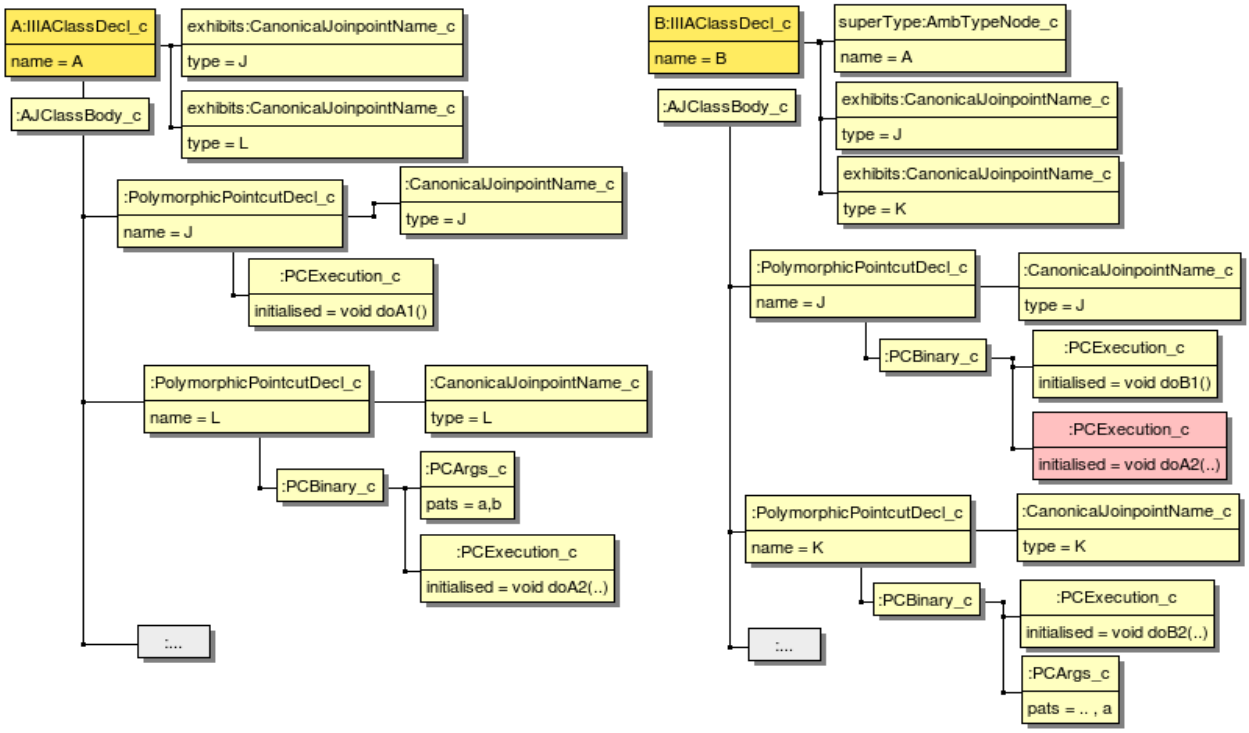

Figure 20: Transformation - PolymorphicPointcutInheriter

In the example the pointcut definition for `J` in class `B` contains a super-pointcut. The transformation replaces the reference to the super-pointcut in class `B` with the pointcut definition for the join point type `J` from class `A`. Figure 20 shows the result.

Pass 11: PolymorphicPointcutFormalsSetter

In this pass the formal parameters of the `PolymorphicPointcutDecl` nodes are set. For this purpose the `JoinpointDecl` node is looked up in the `JoinpointtypeManager` using the `JoinpointName` stored in the `PolymorphicPointcutDecl` node as the key. The `FieldDecl` nodes of the join point type are converted to formal parameters and are added to the pointcut definition.

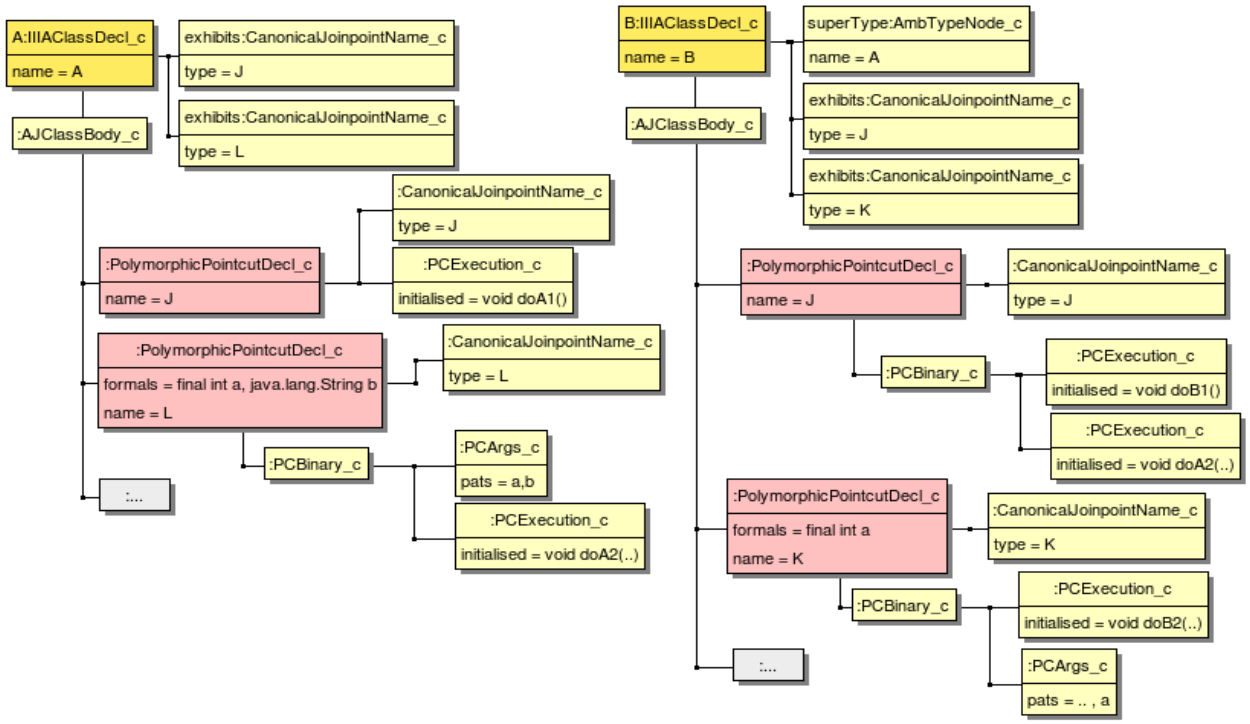

Figure 21: Transformation - PolymorphicPointcutFormalsSetter

Figure 21 shows the setting of pointcut formal parameters in the example. For pointcuts of join point type `J` no formal parameters are set, because join point `J` contains no field declarations. On the other hand pointcuts for join point type `K` the formal parameter `final int a` and for join point type `L` the formal parameters `final int a, java.lang.String b` are set.

Pass 12: ExhibitBlockTransformer

Explicit announcement via an exhibit block is transformed into an anonymous inner class of the exhibited join point type (see Pass 4). The anonymous inner class overrides the method generated by the `JoinpointMethod-Generator` pass and the statement of the block is copied into the method body. The original exhibit bock is replaced by a call of the inner class's method and a pointcut matching the call is added to the class. Thereby it is ensured that advice defined for the join point type is invoked whenever original block is executed. If there already exists a pointcut definition for the block's join point type, the generated pointcut is added to the existing pointcut definition. If there exists no pointcut for the block's join point type, a complete new pointcut is generated and added to the class, which contains the exhibit block.

As illustrated in Figure 22 the exhibit block is transformed into an anonymous inner class of the join point type. A new instance is created, in which the method `_$_do()` is overridden with the code from the `ExhibitBlock` node. Also a new pointcut matching the execution of the overridden join point type method is generated and added to the existing pointcut of `L`.

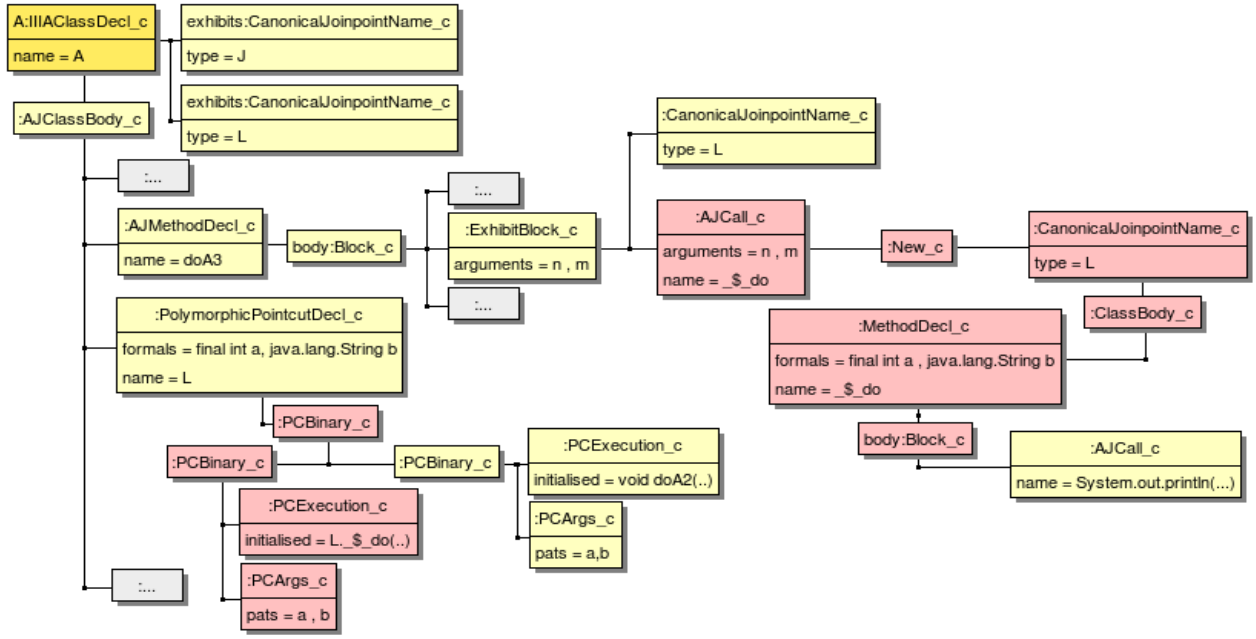

Figure 22: Transformation - ExhibitTransformer

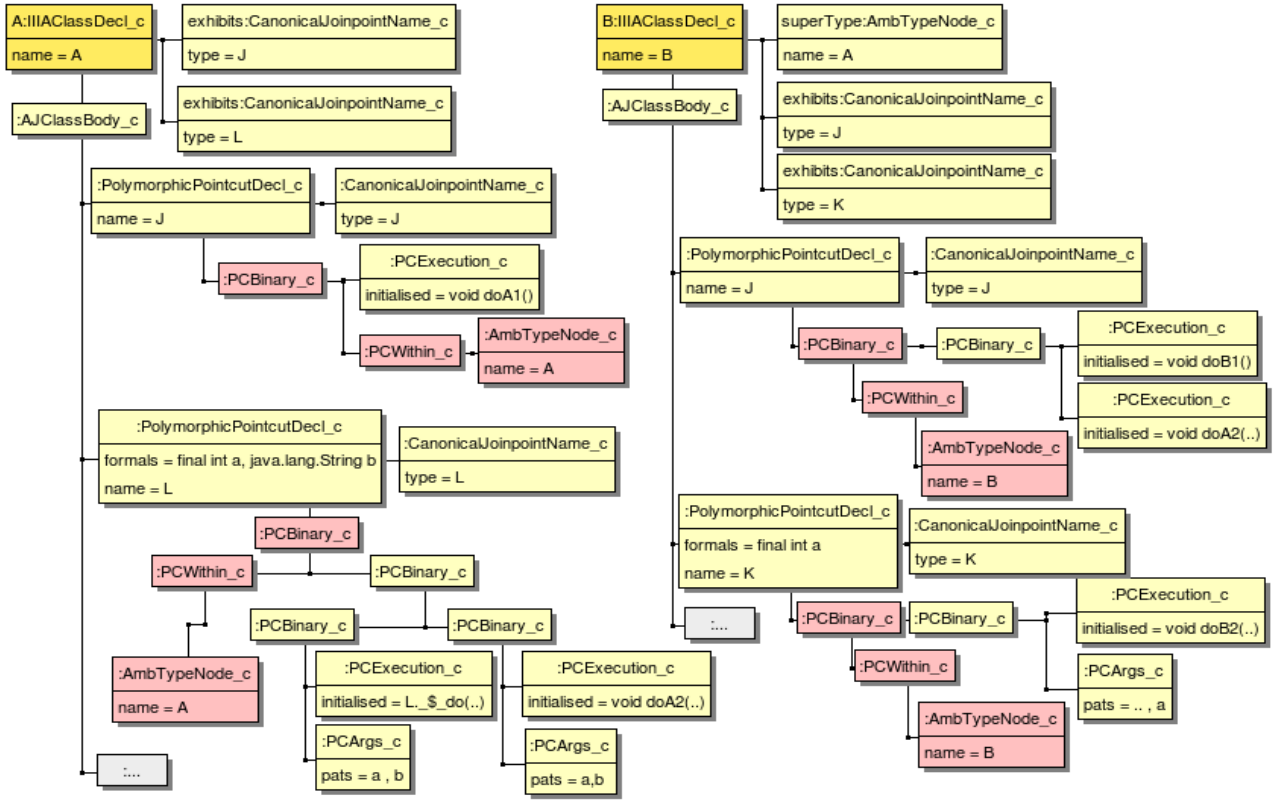

Figure 23: Transformation - PolymorphicPointcutRestrictor

Pass 13: PolymorphicPointcutRestrictor

As already mentioned the class local pointcuts have to be restricted to match only join points within the class they are defined in. So this pass adds a restriction to every `PolymorphicPointcutDecl` node. The AspectJ language offers a proper pointcut definition to achieve this restriction. The lexical pointcut is called `within()` and matches every join point occurring in the type defined by the pattern of the pointcut. In order to restrict the class local pointcut, the original pointcut is conjoined with a `within(<classname>)`, where `<classname>` stands for the class the restricted pointcut is defined in.

Figure 23 shows the AST after the `PolymorphicPointcutRestrictor` restricted the individual `Pointcut-Decl` nodes. By adding the required `within()`-restriction to the individual pointcut nodes.

Pass 14: PolymorphicPointcutSubJoinpointRestrictor

This pass checks every `IIIAClassDecl` node if the class contains a pointcut for a join point type and simultaneously a pointcut for a subtype of this join point type. If this is the case, the pointcut of the supertype must be restricted so that it does not to match when the pointcut of the join point subtype matches. To create the restriction in the pointcut of the join point subtype all variable names within the pointcut are replaced with there types. This is done by using the methods of the interface `NameToTypeConvertablePointcut` which is implemented by the corresponding pointcut nodes (see 9.3.3). After the names of the variable are replaced with their types, the pointcut is negated and conjoined with the original pointcut.

n the example two subtyping-induced pointcut restrictions have to be added to the AST. Class `A` exhibits the join point types `J` and `L`. `L` is a subtype of `J`, so that the pointcut of `J` has to be restricted not to match join points of `L`'s pointcut. After in the pointcut of `L` all variable-names are replaced with their types the pointcut is negated with an `PCNot` node and conjoined with the pointcut of `J` with an `PCBinary` node. The same procedure is applied to the pointcuts for the join point types `J` and `K` in class `B`, because `K` is also a subtype of `J`. The result of the transformation can be seen in Figure 24.

Pass 15: ClassChecker

The `ClassChecker` pass performs semantic checks. In particular, it is checked whether a class contains only pointcuts to join point types that are exhibited by the class. If this check fails, the pass generates an error as this would be inconsistent with the concept of modularity of the IIIA extension. Also, the pass checks if for every exhibited join point type a pointcut exists. If a pointcut is missing a warning is generated to provide information about the missing join point type and the (perhaps) needles exhibition.

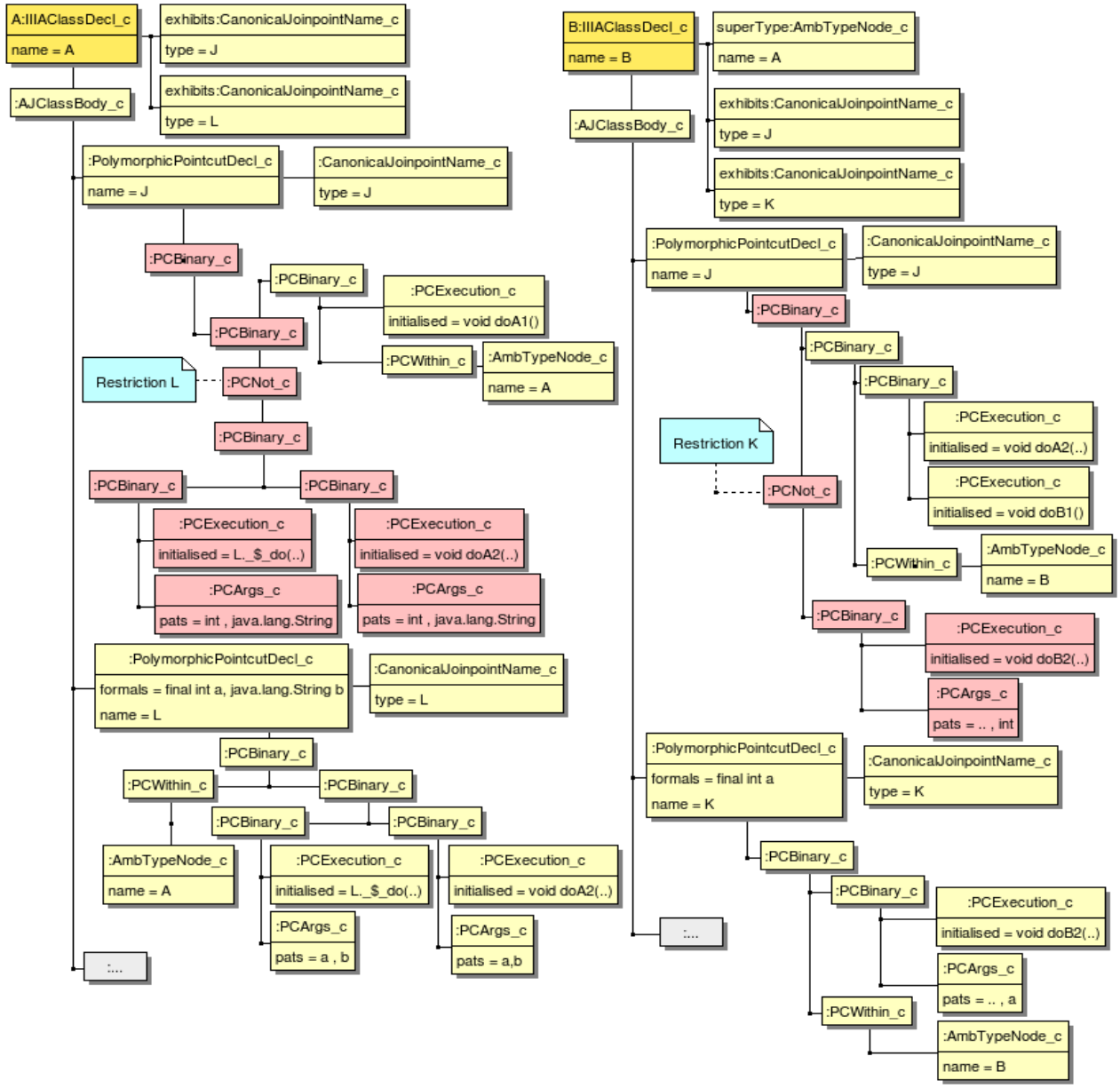

Figure 24: Transformation - PolymorphicPointcutSubjoinpointRestrictor

Pass 16: AspectChecker

The `AspectChecker` does similar checks on aspect side as the `ClassChecker` on class side. It generates an error if an `IIIAAspectDecl` node contains an advice for a join point type which is not advised by the aspect and generates a warning if the aspect advises a join point type, but contains no advice for this join point type.

Pass 16: JoinpointChecker

This pass simply checks, whether a join point type is used. Therefore it is checked, whether the join point type is advised by an aspect and exhibited by a class. If a check fails a warning is generated.

Pass 17: JoinpointSubtypeAdviceGenerator

With this pass the subtyping of join point types on the aspect side is emulated. For this purpose the pass checks if an aspect declares to advise a join point type and a subtype of this join point type simultaneously. If this is the case, it is checked if an advice for the subtype is missing. If this is also the case the advice for the join point supertype is copied and added to the aspect. Before it is added the copy of the advice is modified so that it is associated with the join point subtype. By copying the advice for the join point supertype it is ensured that the aspect's behavior is exactly the same for the join point supertype as for the join point subtype.

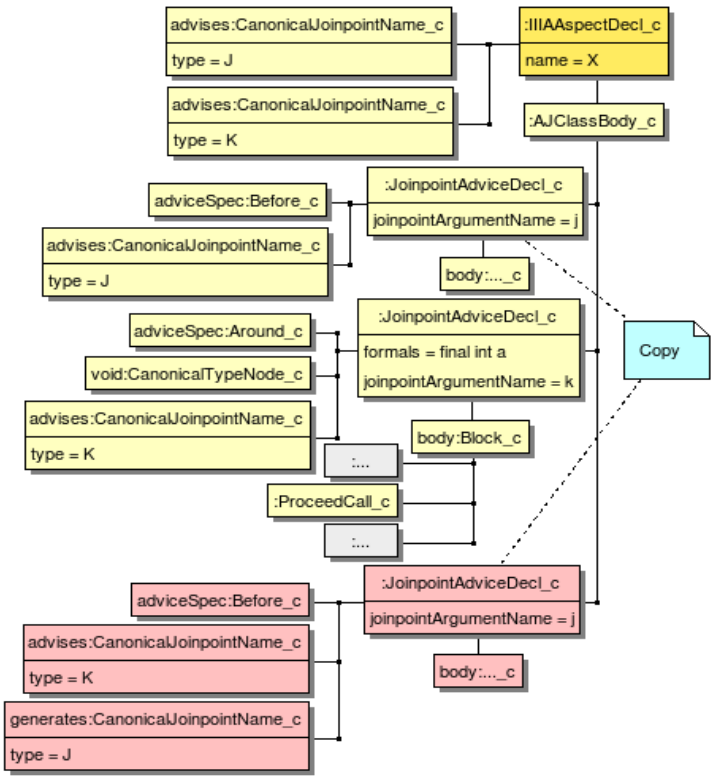

Figure 25: Transformation - JoinpointSubtypeAdviceGenerator

The result of the transformation is Figure 25. The before-advice of join point type J in aspect X has to be copied, because `X` is advising beside `J` the join point type `K`, which is a subtype of `J`, and and concurrently defines no own before-advice for `K`. Therefore the before-advice is copied and added to X. Before the adding it is associated with the join point type `K` and in the copy it is annotated, that a join point type instance of `J` should be generated. This ensures, that in the advice only the context of `J` is available and not the extended context of `K`. This have to be done, because the copied advice is an advice of `J`, even if it is invoked by the occurring of join point type `K`.

Pass 18: PointcutDeclarationGenerator

This pass generates the global pointcut definition of a join point type by visiting every `JoinpointDecl` node. The global pointcut definition is created by disjoining all class local pointcut branches of the join point type. The collection of the pointcut branches is done by exploiting the associations between the `JoinpointDecl` nodes and the

`IIIAClassDecl` nodes, as created by the `ClassExhibitor` pass. The final global pointcut is stored in the `JoinpointDecl` node of the corresponding join point type.

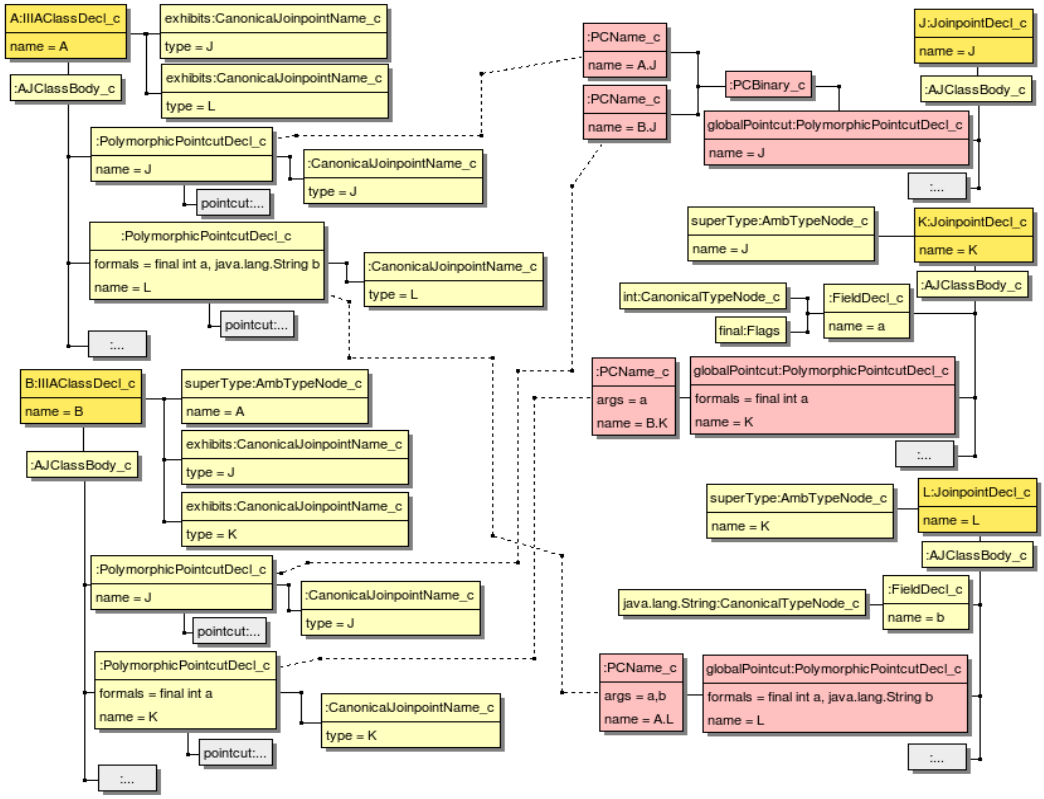

Figure 26: Transformation - PointcutDeclarationGenerator

In Figure 24 it can be seen that for every join point type a `PointcutDecl` node was generated. Each generated node represents the global pointcut for the corresponding join point type. Therefore the global pointcut uses `PCName` nodes for referencing the class local pointcuts.

Pass 19: AdviceTransformer

The `AdviceTransformer` pass visits each advice and connects it with the global pointcut definition of the advice's join point type. Therefore the pass creates a reference to the global pointcut definition of the advice's join point type and adds this reference to the advice. Furthermore the fields of this join point type are converted to formal parameters and are added to the advice in order to capture the context bindings of the global pointcut.

To avoid that in the advice a formal parameter is referenced in the advice accidentally a prefix is added to the names of the formal parameters. This prefix contains keywords, which can not be handled by the parser. If the parser would hit these keywords while expecting a name of a formal parameter a parsing error would be generated. This ensures that the formal parameters can not be used in the source code and that the context of the join point can only be accessed by using the join point type instance (the instance of the join point type is created in the `AdvideJoinpoint-InstanceCreator` pass)

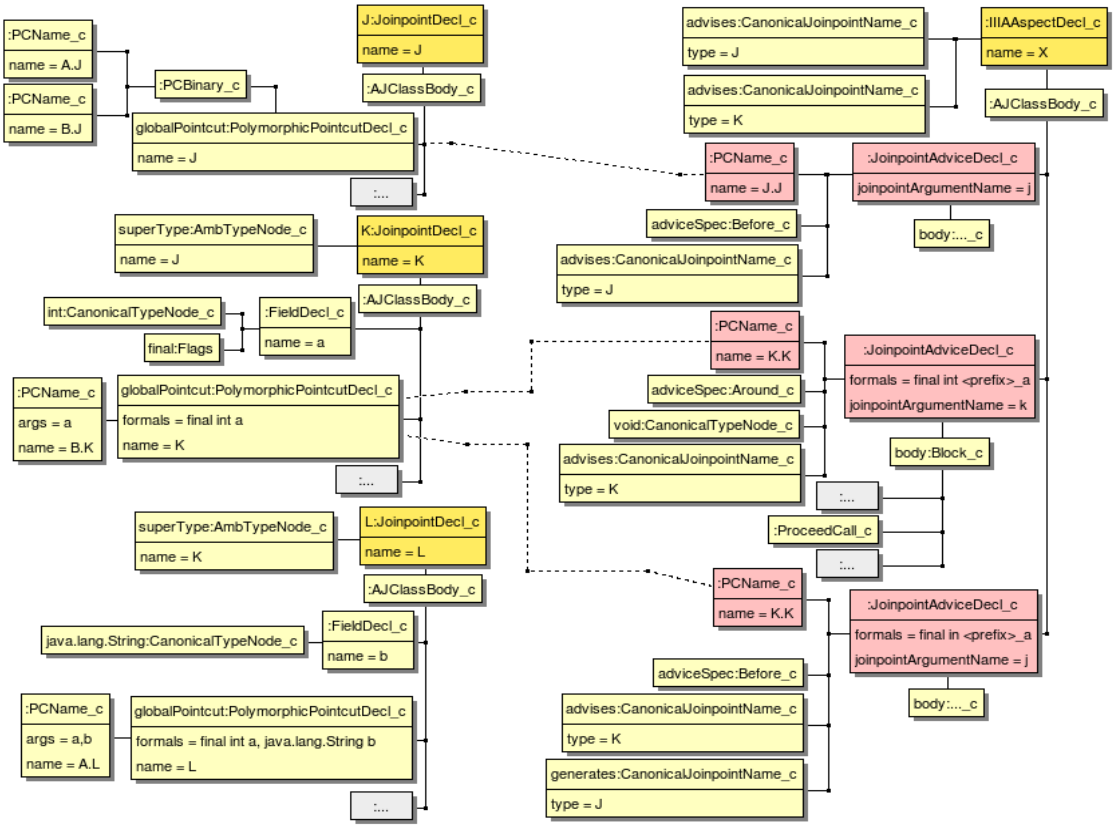

Figure 27: Transformation - AdviceTransformer

Figure 27 shows the transformation of the advice. Each `JoinpointAdviceDecl` node is linked via a `PCName` node with the global pointcut definition of the advice's join point type. Also the formal parameters of the advice are updated with the join point type fields. Because join point type `J` has no fields the corresponding advice has no formal parameters.

Pass 20: UnappliedAdviceRemover

In AspectJ the formal parameters of an advice must correspond with the formal parameters of the advice's pointcut. In IIIA the advice is not linked with its pointcut definition in the source code, but the linked is tied by the `AdviceTransformer` during the compilation. This can cause, that an advice is defined for a certain join point type, but for this join point type there is no (global) pointcut existing[7]. The result is an advice with an empty pointcut definition. If the advice's join point type contains any fields, this fields are added as formal parameters to the advice during the transformation. This produces an advice which formal parameters does not correspond with the formal parameters of its pointcut. When the original AspectJ passes of the AspectBench compiler are executed, this situation would produce compiler errors. Therefore the advices, which are linked with a join point type for which no pointcuts are defined, have to be removed from the AST in order to prevent these errors.

In the example no advice has to be removed.

7 Namely then, when no class local pointcut is defined for the join point type.

Pass 21: AdvideJoinpointInstanceCreator

To make the context of the join point available in an advice a instance of the advice's join point type is created. This is done by using the constructor generated by the `JoinpointConstructorGenerator` pass (see Pass 3). Therefore a constructor call is generated into the advice body. The constructor call is inserted as first statement in the advice's body and uses the actual parameters of the advice as arguments for the call. This ensures that the actual parameters are bound to the join point type fields and are reachable via the join point type instance.

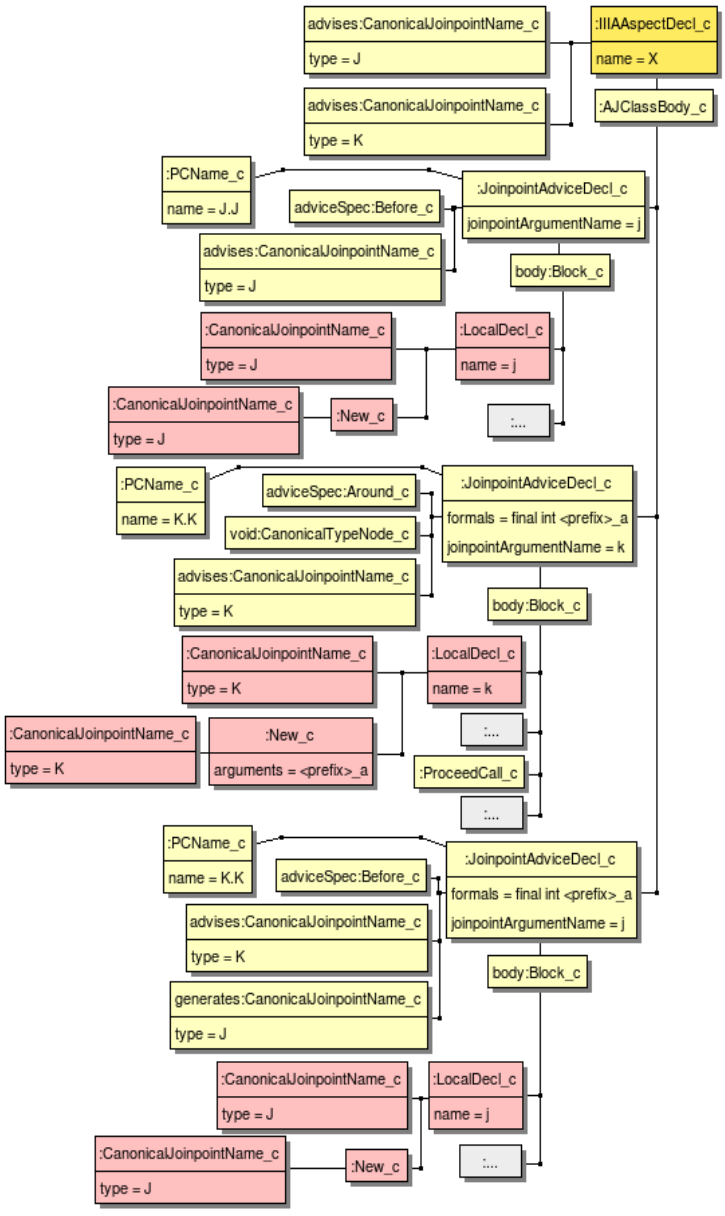

Figure 28: Transformation - AdviceJoinpointInstanceCreator

The instantiation of the join point type is done to provide an object oriented way to handle the join point type. For instance this includes the access to the fields of the join point type. The creation of a join point type instance is nothing different then the definition of local variable within an advice, only that the definition is not included in the source code,

but generated by the compiler. The instantiation can be left out[8] by rewriting each access to a field of the join point type instance to an access to the formal parameters of the advice.

The transformed AST in Figure 28 shows the local declaration of the join point type instances. Therefore for each advice a `LocalDecl` node is generated and added to the body of the advice. As name for the join point type instance the `LocalDecl` node uses the original name of the join point type advice formal parameter, which was extracted by the `AdvideJoinpointFormalExtractor`. Also within the third advice in the aspect a instance of `J` is created, although the advice is linked with join point type `K`. This advice is the one, which was copied by the `Joinpoint-SubtypeAdviceGenerator` pass (Pass 6) in order to simulate the subtyping of join point types. This ensures, that only the context of `J` is available in the copied advice even if the advice is invoked by the occurring of `K`.

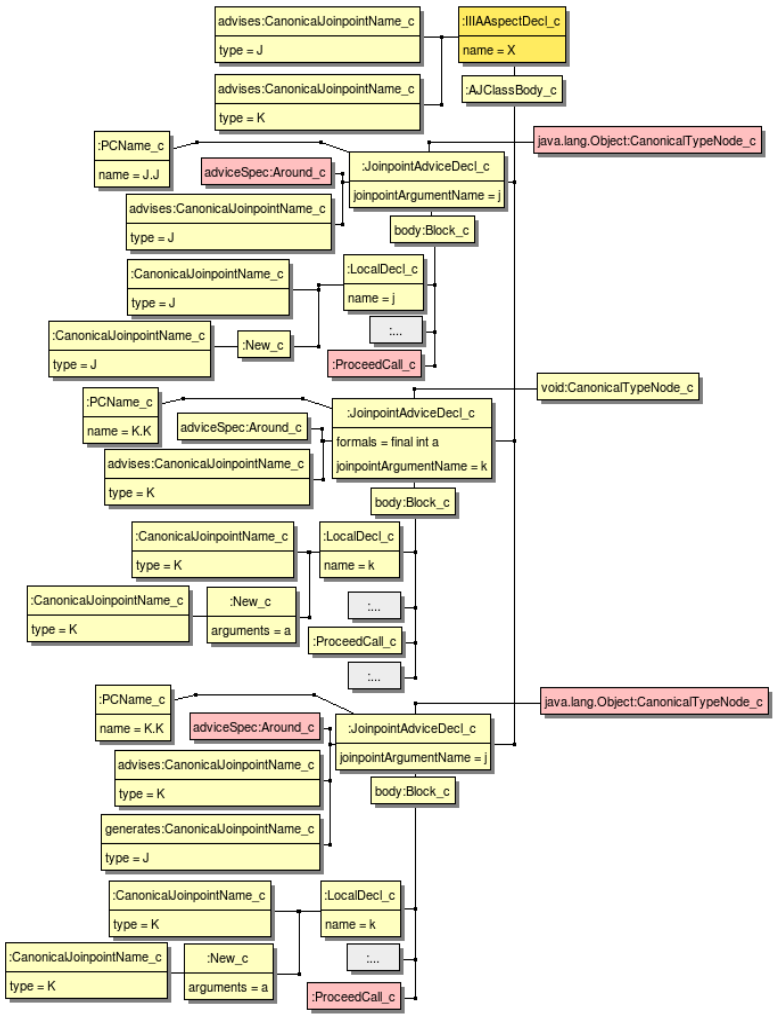

Figure 29: Transformation - BeforeAdviceReplacer

Pass 22: BeforeAdviceReplacer

A before advice is replaced by a around advice for make the changing of the context within the advice possible. A before advice is transformed into an around advice with a return type of `java.lang.Object`. This allows the advice

8 For instance to increase the performance and to decrease the memory consumption.

to return any type, also primitive types (like `int`) or even `void`. After changing the specification of the advice from before into around a proceed call is appended to the advice body. This has to be done in order to simulate the behavior of an before-advice, because by before-advices the original join point code is executed after the advice code. The around advice without the inserted proceed call would replace the original code.

The transformed AST can be found in Figure 29. The advice specifications of the two before advices are replaced by new `Around` nodes and the return type of the advice is changed to `java.lang.Object`. Also a `ProceedCall` node is append to the advice's body.

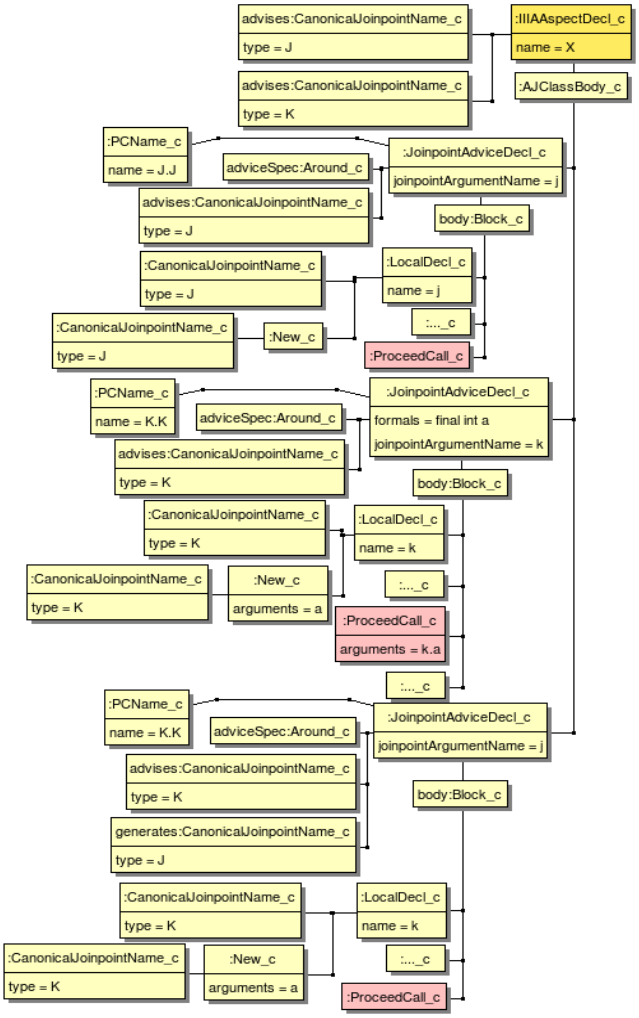

Figure 30: Transformation - ProceedCallArgumentsSetter

Pass 23: ProceedCallArgumentsSetter

Differing from AspectJ the IIIA extension requires that proceed calls have no arguments. Instead, the arguments are inserted into the AST in order to use the AspectJ compiler passes. As the arguments of a proceed call are determined by the advice's formals and in IIIA the advice's formals are represented by the fields of the join point type instance, these fields have to be inserted as arguments of the proceed call.

Figure 30 shows the AST after the proceed arguments are added. For advice of join point type `J` no arguments has to be added, as this join point type has no fields. The proceed in the advice for K contains one argument for the `K`'s field `final int a`.

After all passes have done their transformation the resulting AST is a valid AspectJ AST. The IIIA related information and nodes are removed from the AST or transformed, so that they can be used as AspectJ or Java nodes. This allows the original AspectJ passes and the rest of the AspectBench compiler to generate Java byte code by using the AST as input. Therewith the hole IIIA transformation can be seen as prefix of the hole compilation process.

10. Example of usage of IIIA

In the following subsections the usage of the IIIIA concepts is demonstrated with two examples. The first example is a producer consumer scenario and in the second one some basic business rules for a simple banking account system are implemented. The full code of the examples and further examples can be found on the IIIA-project site [IIIAproject].

10.1. Producer consumer scenario

A typical scenario in operating systems is the producer consumer problem. This scenario can be sketched as a producer produces items and a consumer consumes the items. The communication between the producer and the consumer is realized by buffer. A problem occurs when the producer wants to store a new item in the buffer and the buffer is full or when the consumer wants to consume an item and the buffer is empty.

With IIIA this scenario can be handled very simply and understandably. First, there is a producer class whose instances produces the items. In the example these items are only wrappers around random integer values, but can be replaced by any other objects. The counterpart to the producer is the consumer. It takes an item and consumes it. In addition, two types of join points are defined in the program. Instances of the first type occur when a new consumer is created, those of the second are generated when an item is produced.

The link between a producer and a consumer is an aspect named Dispatcher. The dispatcher's advice is triggered by the two join point types. The new consumer event prompts the dispatcher to store a reference to the new consumer in a list. When the dispatcher's advice is invoked by the production of a new item, this new item is handed over to the dispatcher via a join point type instance. Then the value of the item is checked and, when it is not valid (in the example the value “0” is invalid), rejected. If the value passes the check, the dispatcher starts to find an idle consumer (a consumer which is not consuming at the moment). If there is an idle one in the list, the item is passed to the consumer. If there is no idle consumer in the list, the dispatcher is suspended for a short time and after the pause retries to find an idle consumer in the list.

A overview of the scenario can be seen in Figure 31.

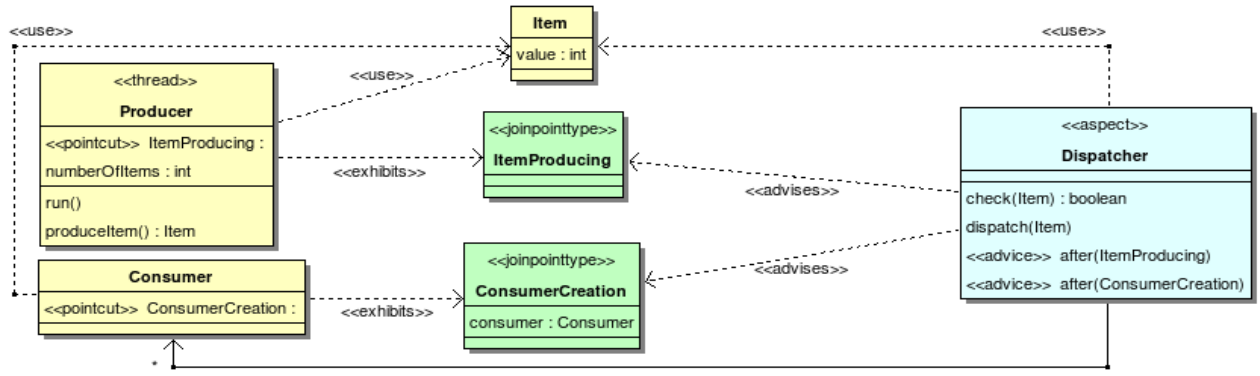

Figure 31: Overview over producers/consumers scenario

The implementation of this scenario works with two join point types. The first is called `ItemProducing` and should occur whenever an item is produced.. The second join point type is named `ConsumerCreation` and arises whenever a consumer is created. This will be used by the dispatcher to build up a list of consumers, for which purpose a field of type `Consumer` is defined within the join point type. The dispatcher then searches the list for a consumer to which an item can be dispatched. Listing 9 introduces the two join point types.

```
public joinpointtype ItemProducing {}

public joinpointtype ConsumerCreation {
   Consumer consumer;
}
```

Listing 9 : Producer-consumer join point types

Listing 10 shows the definition of item which is produced and consumed. As already mentioned it is a simple wrapper around an integer value, but the type of the value is replaceable.

```
public class Item {
   public int value = 0;
}
```

Listing 10 : Item class

The producer's implementation can be found in Listing 11. `Producer` implements the `java.lang.Runnable` interface in order to run the item production in an own thread. The method `run()` simply produces a certain amount of items. Furthermore the class exhibits the join point type `ItemProducing`. To satisfy the requirements of IIIA the class has to provide a pointcut for the exhibited join point type. In our example the pointcut for the `ItemProducing` type matches the execution of the method `produceItem()`.

```
public class Producer implements Runnable exhibits ItemProducing {

   pointcut ItemProducing : execution(Item produceItem());

   private int numberOfItems = 0;

   public Producer(int numberOfItems) {
      this.numberOfItems = numberOfItems;
   }
```

```
  public void run() {
     System.out.println("[producer]\tstart producing "+numberOfItems+" items");
     for(int i = 0; i < numberOfItems; i++) {
        Item it = produceItem();
        System.out.println("[producer] item "+i+" : "+it.value);
        ...
     }
     System.out.println("[producer] stop producing");
  }

  private Item produceItem() { ...}
}
```

Listing 11 : Producer class

Listing 12 shows the implementation of `Consumer`. The consumer is simple Java class exhibiting the join point type `ConsumerCreation`. The pointcut of this join point type matches the constructor of the class and exposes the just created `Consumer` instance. The real consumption of an `Item` instance is done in the method named `consume(Item)`.

```
public class Consumer exhibits ConsumerCreation {
  pointcut ConsumerCreation : execution(new(..)) && this(consumer);

  private java.util.Random random = new java.util.Random();
  private boolean working = false;

  private String name;

  public Consumer(String n) {
     name = n;
  }

  public String getName() {return name;}

  public void consume(Item it) {

     working = true;
     System.out.println("["+name+"] consuming "+it.value);
     ...
     working = false;

  }

  public boolean isWorking() {return working;}

}
```

Listing 12 : Consumer class

Listing 13 shows the implementation of `Dispatcher`. This aspect is responsible for dispatching the produced items to the consumers. Therefore it advises the two join point types `ItemProducing` and `ConsumerCreation`. For both join point types the aspect offers an advice. The first advice is invoked, when a `Consumer` instance is being created. To build up a list of consumers the advice stores the instance of `Consumer` in the list `consumers` within the aspect. The second advice is invoked, when a new item is produced. Firstly the produced item is checks by the method `check(Item)`. If th item passes the check, the `Dispatcher` searches in the list of consumers for an idle one. This is done in the method `dispatch(Item)` which starts for the dispatching an own thread so that the producer is not

blocked and can continue with the production of items. If no idle consumer the dispatching-thread is suspended and after the pause the search for on idle consumer is started again.

```
public aspect Dispatcher advises ItemProducing, ConsumerCreation {

  private java.util.List consumers = new java.util.ArrayList();

  after(ConsumerCreation creation) returning {
     System.out.println("[dispatcher] add new consumer: "+creation.consumer.getName());
     consumers.add(creation.consumer);
  }
     after(ItemProducing producing) returning(Item it) {
     System.out.println("[dispatch] receive item : "+it.value);
     boolean check = check(it);
     if(check) {
        dispatch(it);
     } else {
        System.out.println("[checking] invalid value "+it.value+"! item rejected");
     }
  }
     private boolean check(Item it) {...}

  private void dispatch(final Item it) {
     Runnable r = new Runnable() {
        public void run() {
           System.out.println("[dispatch] search for idle consumer");

           boolean searching = true;
           while(searching) {
              for(java.util.Iterator i = consumers.iterator(); i.hasNext();) {
                 Consumer c = (Consumer) i.next();
                 if(!c.isWorking()) {
                    System.out.println("[dispatch] "+c.getName()+" is idle ...");
                    c.consume(it);
                    searching = false;
                    break;
                 }
                 ...
              try {
                 Thread.sleep(1000);
              } catch(InterruptedException e) {
                 System.err.println(e);
              }
           }
        }
     };
     Thread th = new Thread(r);
     th.start();
  }
}
```

Listing 13 : Dispatcher aspect

An exemplary output may look like in Listing 14.

```
[dispatcher] add new consumer: Consumer 1
[dispatcher] add new consumer: Consumer 2
[dispatcher] add new consumer: Consumer 3
[producer]      start producing 20 items
[dispatch] receive item : 2
[producer] item 0 : 2
[dispatch] search for idle consumer
[dispatch] Consumer 1 is idle ...
[Consumer 1] consuming 2
```

```
[Consumer 1] sleeping for 2527
[dispatch] receive item : 2
[producer] item 1 : 2
[dispatch] search for idle consumer
[dispatch] Consumer 1 is busy
[dispatch] Consumer 2 is idle ...
[Consumer 2] consuming 2
[Consumer 2] sleeping for 2412
[dispatch] receive item : 3
...
[dispatch] receive item : 0
[checking] invalid value 0! item rejected
[producer] item 4 : 0
[Consumer 3] ready for new Items
[dispatch] receive item : 4
[producer] item 5 : 4
[dispatch] search for idle consumer
...
```

Listing 14 : Output of producer-consumer-scenario

10.2. Business rules

An example from [AJIA2003] illustrates the use of AspectJ for implementing business rules. Business rules are thought to be crosscutting concerns because they influence not only a single functionality in the software system but rather affect many business transactions implemented in the software system. In the example of [AJIA2003], the scenario of a simple banking system is chosen. The system allows to debit and credit an account and to transfer an amount of money from one account to another.

In this example also a simple bank system is implemented with the same functionality: debit and credit an account as well as the transfer from one account to another. Furthermore the following business rules are implemented:

- dynamic credit check

  If an account is debited, it is checked, whether the total capital of the customer covers the debit

- charging an account with transfer fees

  On every account transfer the transferring account is charged with a transfer fee. Exclusion is, if the owner of both accounts is the same. Also the transfer fee depends on the amount of the transfer.

At first the basic classes are defined, which covers the core concerns of the banking system. These are classes for a `Customer`, an `Account` and an `AccountTransferSystem`. The implementation of the `Customer` is kept very simple. It defines that a customer has a name and can own many accounts. The `Account` class is responsible for managing the balance of an account. To identify an account, it contains an account-number. For crediting and debiting the class offers corresponding methods. If an account is credited and the balance is unscientific to cover the credit, a `InsufficientBalanceException` is thrown. The `AccountTransferSystem` transfers a certain amount from one account to another one.

The business rules are implemented in aspects. One aspect, called `DebitingRules`, is responsible for the dynamic credit check and another aspect, called `TransferRules`, implements the calculating and charging of the transfer fees

during a transfer. In order to invoke the aspect's advices, join point types are required. For this reason the implementation of the business rules uses three join point types: `AccountAction`, `Debiting` and `Account-Transfering`. The join point type `Debiting` is exhibited by the class `Account` which defines a pointcut for the join point type matching the `debit()`-method of `Account`. Furthermore the aspect `DebitingRules` advises the `Debiting` and contains a before advice which checks, whether the total capital of the customer covers the debit. The join point type `AccountTransfering` is exhibited by `AccountTransferSystem`. This class defines a pointcut for the join point type which matches the `transfer()`-method. On aspect side, `AccountTransfering` is advised by `TransferRules` which defines an around advice for the join point type which computes the transfer fee and charges the account initiating the transfer with this computed fee.

An overview over the hole scenario is given in Figure 32.

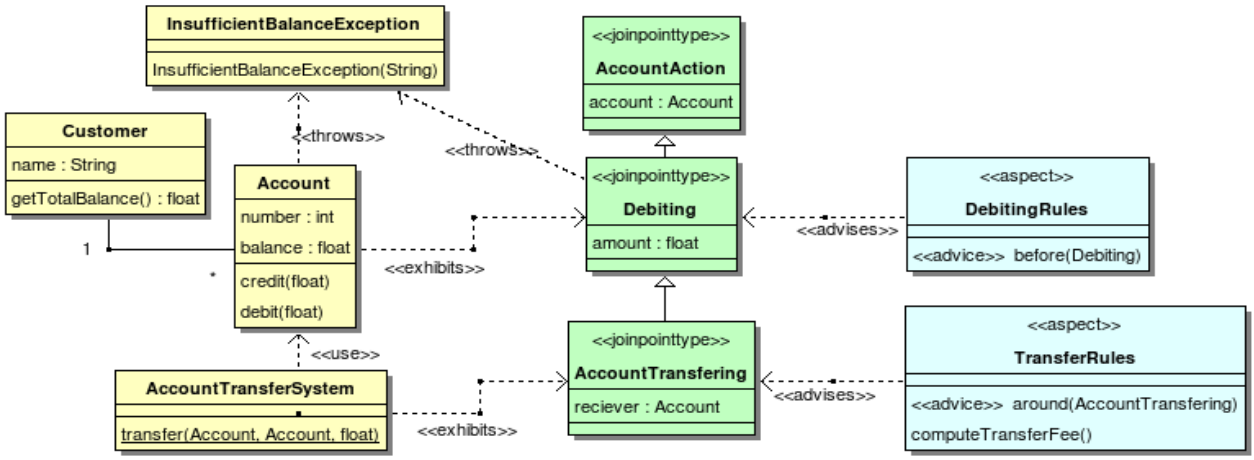

Figure 32: Overview over business rules scenario

In the following the implementation is described in more detail. Listing 15 shows the join point types.

```
public joinpointtype AccountAction {
   Account account;
}

public joinpointtype Debiting extends AccountAction{
   float amount;
}

public joinpointtype AccountTransfering extends Debiting {
   Account reciever;
}
```

Listing 15 : Join point types in business rules example

Listing 16 contains the implementation of the `Customer` class.

```
public class Customer {
   private Map accounts = new HashMap();
   private String name;

   public Customer(String name) {this.name = name;}
```

```
   public String getName() {return name;}
   public void addAccount(Account account) {...}
   public void removeAccount(Account account) {...}
   public Account getAccount(int number) {...}
   public float getTotalBalance() {...}

}
```

Listing 16 : Customer class

The implementation of an `Account` can be found in Listing 17. This class exhibits the join point type `Debiting`. Therefore the class offers a pointcut matching the `debit()` method. Furthermore the pointcut exposes the amount of the debit and the instance of `Account`, which is debited.

```
public class Account exhibits Debiting{

   pointcut Debiting:
      execution(void debit(..)) && args(amount) && this(account);

   public Account(int number) {...}
   public Account(int number, float balance) {...}
   public int getAccountNumber() {...}
   protected void setBalance(float balance) {...}
   public float getBalance() {...}

   public void credit(float amount) {setBalance(getBalance() + amount);}

   public void debit(float amount) {setBalance(getBalance() - amount);}
   public Customer getOwner() {...}
   public void setOwner(Customer new_owner) {...}
}
```

Listing 17 : Account class

Next, the `AccountTransferSystem` is introduced in Listing 18. This class offers a simple static method, which transfers a certain amount from one account to another. For including the transfer rules of the software system the class exhibits the join point type `AccountTransfering` and offers a pointcut for it. This pointcut matches the execution of the `transfer()`-method, which causes the invoking of the advice of `AccountTransfering`.

```
public class AccountTransferSystem exhibits AccountTransfering{
   pointcut AccountTransfering :
      execution(void transfer(Account, Account, float))
      && args(account, receiver, amount);
   public static void transfer(Account from, Account to, float amount) {
      from.debit(amount);
      to.credit(amount);
   }
}
```

Listing 18 : AccountTransfer class

Now, the aspects, which implements the rules, are described. Listing 19 starts with the aspect `DebitingRules`. The aspect advises the join point type `Debiting` and defines an advice for it. The advice simply checks, if the customer's total capital covers the amount of the debit. If the customer capital does not cover the amount an `Insufficient-BalanceException` is thrown.

```
public aspect DebitingRules advises Debiting {

   before(Debiting debit) {
      log("Debiting ["+debit.amount+"] from : "+debit.account);
      // Get total money of customer
      float total = debit.account.getOwner().getTotalBalance();
      if(total < debit.amount) {
         throw new InsufficientBalanceException("Total Balance of "+
         debit.account.getOwner().getName()+" is with "+total+" "+
         "unsufficient for debiting "+debit.amount);
      }
   }
}
```

Listing 19 : DebitingRules aspect

The next listing, Listing 20, shows the aspect `TransferRules`. This aspect computes the transfer fee and charges the account, from which the money is transfered, with this computed fee. Therefore the aspect advises the join point type `AccountTransfering` and offers an advice for it. This around advice calls the aspect's method `computeTransferFee()`, invokes the original join point by an proceed() and charges afterwards the account with the fee.

```
public aspect TransferRules advises AccountTransfering {
   void around(AccountTransfering transfer) {
      float transfer_fee = computeTransferFee(transfer.account,
transfer.receiver,transfer.amount);
      try {
         proceed();
         if(transfer_fee > 0) {
            transfer.account.debit(transfer_fee);
         }
      } catch(Exception e) {System.out.println(e);}
   }
   private float computeTransferFee(Account account, Account receiver, float amount)
{...}
   private static void log(Object msg) {...}
}
```

Listing 20 : Transfer rules aspect

11. Summary and conclusion

11.1. Summary of contribution

Being an AspectJ based extension for Java the approach of implicit invocation with implicit announcement (IIIA) integrates join point types in Java and therewith results in the introduction of certain new language elements. These new elements were mapped to existing Java/AspectJ language elements and a transformation was developed that is able to accomplish this mapping.

Based on this transformation the compiler supporting IIIA was implemented by creating an extension for the AspectBench compiler framework (abc). The implemented compiler extension consists primarily of a parser, several new nodes of the abstract syntax tree (AST) and several compiler passes. The parser extends and adapts the AspectJ

language grammar in order to support the IIIA syntax. Furthermore the parser uses the new nodes of the compiler extension for creating an abstract syntax tree during the parsing of IIIA-programs. Thereby the new AST nodes contains additional information, which are required by the compiler passes of the compiler extension. These compiler passes are responsible for the transformation of the AST into a pure and valid AspectJ-AST, so that the AspectJ functionality of the AspectBench compiler can finish the compilation and can produce Java byte code from the AspectJ-AST. The implementation of the compiler and with it the approach of IIIA was tested on some examples that demonstrate the promised modularity of the approach.

11.2. Conclusions

After having applied IIIA in a couple of examples a statement concerning usage and usability of the compiler and the supported approach of IIIA can be made as follows:

Compared to AspectJ programs, pointcut definitions in IIIA programs become smaller and less complex and therefore better readable. At the same time, they are scattered over the exhibiting classes. This is the result of splitting the pointcut into class local pointcuts and distributing them to the classes they match in. Depending on standpoint this is no disadvantage, since it reduces the obliviousness and thus the anti-modularity AspectJ is often criticized for,

References

[abc] : abc-Team "The AspectBench Compiler for AspectJ" , http://abc.comlab.ox.ac.uk/, last visit: 02.07.2007

[abc2005] : Pavel Avgustinov, Aske Simon Christensen , Laurie Hendren, Sascha Kuzins, Jennifer Lhotak, Ondfej Lhotak, Oege de Moor, Damien Sereni , Ganesh Sittampalam, Julian Tibble "abc : An extensible AspectJ compiler" , http://abc.comlab.ox.ac.uk/documents/taosd2005.pdf, last visit: 02.07.2007

[AJIA2003] : Ramnivas Laddad "AspectJ in Action", 2003, 1-93-0110-93-5

[AJIA2003] : Ramnivas Laddad "AspectJ in Action", 2003, ISBN 1-930110-93-6

[CUP] : CUP-Team "CUP - LALR Parser Generator in Java" , http://www2.cs.tum.edu/projects/cup/, last visit: 02.07.2007

[IIIAproject] : Friedrich Steimann, Thomas Pawlitzki "Modular Programming with Join Point Types and Polymorphic Pointcuts" , http://www.fernuni-hagen.de/ps/prjs/IIIA/, last visit: 02.07.2007

[Polyglot] : Andrew Myers "Polyglot - A compiler front end framework forbuilding Java language extensions" , http://www.cs.cornell.edu/Projects/polyglot/, last visit: 02.07.2007

[PolyglotDoc] : Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers "Polyglot: An Extensible Compiler Framework for Java" , http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cs/TR2002-1883, last visit: 02.07.2007

[PPG] : Michael Brukman, Andrew C. Myers "PPG - A parser generator for extensible grammars" , http://www.cs.cornell.edu/Projects/polyglot/ppg.html, last visit: 02.07.2007

[Soot] : Patrick Lam, Feng Qian, Ondrej Lhotak "Soot: a Java Optimization Framework" , http://www.sable.mcgill.ca/soot/, last visit: 02.07.2007

Ehrenwörtliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht.

Köln, den 02.07.2007 Thomas Pawlitzki