Join Point Interfaces for Safe and Flexible Decoupling of Aspects

ERIC BODDEN, Technische Universität Darmstadt
ÉRIC TANTER and MILTON INOSTROZA, University of Chile

In current aspect-oriented systems, aspects usually carry, through their pointcuts, explicit references to the base code. Those references are fragile and hinder important software engineering properties such as modular reasoning and independent evolution of aspects and base code. In this work, we introduce a novel abstraction called Join Point Interface, which, by design, aids modular reasoning and independent evolution by decoupling aspects from base code and by providing a modular type-checking algorithm. Join point interfaces can be used both with implicit announcement through pointcuts, and with explicit announcement, using closure join points. Join point interfaces further offer polymorphic dispatch on join points, with an advice-dispatch semantics akin to multimethods. To support flexible join point matching, we incorporate into our language an earlier proposal for generic advice, and introduce a mechanism for controlled global quantification. We motivate each language feature in detail, showing that it is necessary to obtain a language design that is both type safe and flexible enough to support typical aspect-oriented programming idioms. We have implemented join point interfaces as an open-source extension to AspectJ. A case study on existing aspect-oriented programs supports our design, and in particular shows the necessity of both generic interfaces and some mechanism for global quantification.

Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms: Design, Languages

Additional Key Words and Phrases: Aspect-oriented programming, modularity, typing, interfaces, implicit announcement, explicit announcement, join point polymorphism, advice dispatch

ACM Reference Format:
Eric Bodden, Éric Tanter, and Milton Inostroza. 2014. Join point interfaces for safe and flexible decoupling of aspects. ACM Trans. Softw. Eng. Methodol. 23, 1, Article 7 (February 2014), 41 pages.
DOI: http://dx.doi.org/10.1145/2559933

1. INTRODUCTION

Aspect-Oriented Programming (AOP) [Kiczales et al. 1997] is a paradigm that aims at enhancing modularity of software by locally handling crosscutting concerns that would otherwise be scattered in different parts of a given system. The emblematic language mechanism of aspect-oriented programming languages is pointcuts and advice, where *pointcuts* are predicates that denote *join points* in the execution of a program where *advice* is executed. The AspectJ programming language [Kiczales et al.

E. Bodden is supported by the German Research Foundation (DFG) within the Emmy Noether Group RUNSECURE, by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE and by the Hessian LOEWE excellence initiative within CASED. É. Tanter is partially funded by FONDECYT Project 1110051.
Authors' addresses: E. Bodden (corresponding author), Secure Software Engineering Group, EC SPRIDE, Mornewegstr. 30, 64293 Darmstadt, Germany; email: eric.bodden@gmail.com; É. Tanter and M. Inostroza, Computer Science Department, University of Chile, Blanco Encalada 2120, Santiago, Chile.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2014 ACM 1049-331X/2014/02-ART7 $15.00
DOI: http://dx.doi.org/10.1145/2559933

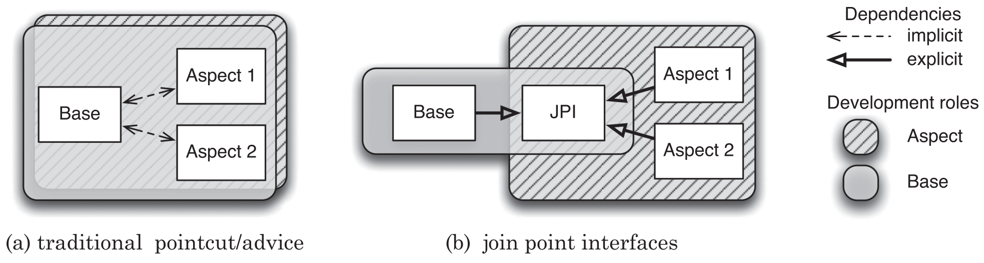


Fig. 1. Dependencies with traditional pointcut/advice AOP (a) and with join point interfaces (b).

2001] is the most widely used language for this style of AOP. Aspects allow crosscutting concerns to be better localized, easing understanding. However, the implicit invocation mechanism introduces subtle implicit dependencies between base code and aspects that compromise a number of properties usually associated with modularity, such as separate development. In fact, it is usually not possible to reason about an aspect or an advised module in isolation.

As we show in Figure 1(a), aspects with pointcuts and advice as in AspectJ cause implicit dependencies between aspects and base code. This is because an aspect contains direct textual references to the base code via its pointcuts. These implicit dependencies, denoted by the dashed arrows in the figure, make programs fragile, hinder aspect evolution and reuse, and compromise separate development. Changes in the base code can unwittingly render aspects ineffective or cause spurious advice applications. Conversely, a change in a pointcut definition may cause parts of the base program to be advised without notice, breaking some implicit assumptions. This problem is known as the *fragile pointcut problem* [Gybels and Brichau 2003; Stoerzer and Graf 2005]. Effectively this means that both developers that maintain aspects and developers that maintain base code must usually have some global knowledge about the software system, for example, knowing the aspects that are defined in order to determine if they affect a given module. The fact that independent development is compromised this way is particularly worrying considering that programming aspects requires a high level of expertise, and is hence likely to be done by specialized programmers, leading to different development roles. Therefore, to be widely adopted, AOP is in need of mechanisms to support separate development in a well-defined manner.

The preceding issues have been identified early on [Gudmundson and Kiczales 2001] and have triggered a rich discussion in the community [Kiczales and Mezini 2005; Steimann 2006]. In particular, several proposals have been developed to enhance the potential for modular reasoning by introducing a notion of *interface* between aspects and advised code (e.g., [Gudmundson and Kiczales 2001; Aldrich 2005; Sullivan et al. 2010; Steimann et al. 2010]). Notably, crosscutting programming interfaces (XPIs) provide programmers with a design regimen to structure aspect-oriented programs such that they can be more easily maintained [Sullivan et al. 2010]. XPIs aid reasoning and maintainability by refactoring elements such as pointcuts, which link aspects to base code, into a common interface. Programmers can then enrich this interface with semantic information about the advising relationship at will. Steimann et al. [2010] built on this idea by introducing a language mechanism for denoting such interfaces through a novel notion of Join Point Types (JPTs). Join points are denoted through nominal types, allowing aspects to be decoupled from base code syntactically. Further, it allows join point subtyping, which increases the potential for reusing advice over multiple related join point types.

However, we show that none of the existing approaches supports *safe* modular type checking of aspects and base code. Modular type checking is an important precursor for modular reasoning. Without the ability to soundly type-check aspects and base code modularly, one can obtain subtle runtime errors after composing aspects with other modules, although each part taken separately is considered type-correct. XPIs, as a design regimen, use the syntax and type-checking semantics of plain AspectJ, which, as pointed out earlier [Jagadeesan et al. 2006; De Fraine et al. 2008; Bodden 2011], has an unsound type system; we give several examples later in the article. Join point types by Steimann et al. do make an important step towards modular type checking. As we show in this work, however, the syntax and static semantics of JPTs is insufficient to ensure safe modular type checking, and the dynamic semantics can lead to peculiarities such as a single advice executing several times at the same join point.

In this work, we combine important ideas of existing earlier work with novel insights obtained through our own research, to arrive at a language design that enables safe modular type checking for aspects. Our solution, called *Join Point Interfaces* (JPIs), consists of type-based contracts that decouple aspects from advised code (Figure 1(b)). Similar to JPTs, JPIs support a programming methodology where aspects only specify the types of join points they advise, but do not comprise any pointcuts. It is the responsibility of the programmer maintaining the advised code to specify which join points are exposed, and of which type. Quantification is now local, restricted to a given class. In addition to implicit announcement through pointcuts that quantify over join points, JPIs are integrated with Closure Join Points [Bodden 2011], a mechanism for explicit event announcement. With closure join points the dependencies become explicit, and therefore simplify reasoning.

Figure 1(b) gives an overview of our design. The figure only contains solid arrows: both aspects and base code explicitly refer to JPIs, without implicit dependencies between them. This supports a form of modular reasoning: since JPIs serve as a type-safe contract, at least as far as the type system is concerned, the base-code developer can reason about base code by just inspecting base code and the JPIs that it explicitly references; similarly, the aspect programmer can reason locally about aspects and the referenced JPIs, without having to worry about base code. The static semantics of JPIs gives the strong guarantee that programmers can safely compose aspects and advised modules, even when they were developed separately.

Like JPTs, join point interfaces support a notion of subtyping, which helps in structuring and managing the complexity of the space of events-of-interest to aspects. Subtyping of JPIs supports join point polymorphism and advice overriding. We introduce a novel semantics of advice dispatch that avoids the join point duplication semantics of JPTs, which can surprisingly yield to one advice being applied several times at the same program execution point.

As we will explain in more detail later, safe modular type checking is at odds with the flexible quantification mechanism that AspectJ offers. Wide quantification over join points hinders both modular reasoning and type checking: if pointcuts may match many types of join points[1] then a type system must be able to soundly but flexibly abstract from those types. To address this issue, we introduce into our design the idea of *generic* JPIs, which use type parameters as originally proposed by Jagadeesan et al. [2006]. To allow a controlled way of wide join point quantification, we introduce a mechanism called *global* JPIs, which use a refinement mechanism to balance the design trade-off between unrestricted quantification as in AspectJ and class-local quantification as in JPTs. As we will discuss further, global JPIs are a pragmatic solution, which gives up

[1]Such as join points returning a value or not, or throwing an exception or not, or exposing different numbers of arguments. . .

some of the explicitness depicted in Figure 1(b), but restores the flexible quantification mechanism from AspectJ in a controlled manner.

We have implemented join point interfaces as an extension to the AspectBench Compiler [Avgustinov et al. 2005], and have evaluated our approach on a number of existing AspectJ software systems. The evaluation shows that the language design is practical for those systems, but also that each language feature is indeed necessary to obtain both type safety and flexibility. In particular, any type-safe language design that wishes to support flexible global quantification must support generic types. The JPI project Web page contains a detailed documentation of the language extension, download links to the open-source implementation, as well as all subject programs used in the case study: `http://bodden.de/jpi/`

Compared to existing work, to the best of our knowledge, this work presents the following original contributions:

—an interface mechanism for safe modular type checking in presence of both implicit and explicit announcement,
—a type-safe static semantics for generic type variables in place of checked exceptions,
—a novel runtime semantics for advice dispatch based on join point polymorphism,
—a mechanism to support highly crosscutting aspects in a controlled manner,
—a full open-source implementation of the preceding as an AspectJ extension, and
—an extensive case study of existing aspect-oriented projects, used to assess the proposed design.

The notion of JPIs was first presented in the New Ideas track of ESEC/FSE [Inostroza et al. 2011]. This work is the result of the development and maturation of that idea; syntax and semantics have evolved, and both implementation and evaluation are completely new; generic JPIs and global JPIs are also both new, since their introduction was motivated by the case study.

Outline. The remainder of this article is structured as follows. In the next section we use a range of examples to show why type checking AspectJ is hard, and why it is generally at odds with the flexibility that AspectJ offers. We further show why existing approaches do not support safe and flexible decoupling of aspects. In Section 3, we illustrate how join point interfaces build upon previous work to address these shortcomings. Section 4 explains our type system and the dynamic semantics in full generality. We discuss our implementation in Section 5, and present the results of a case study in Section 6. Section 7 discusses related work and Section 8 concludes.

2. EXISTING APPROACHES

In this section we motivate our approach by first reviewing the benefits and pitfalls of the state-of-the-practice in pointcut-advice AOP, namely AspectJ. We then explain different proposals to achieve a better decoupling of aspects and base code, and focus especially on the proposal of Steimann et al. [2010], which serves as a starting point to this work. We explain why these different approaches fall short of achieving both safe and flexible decoupling of aspects.

2.1. AspectJ

To illustrate the benefits and pitfalls of AspectJ, we use a running example based on an e-commerce system, directly inspired from related work [Steimann et al. 2010]. Consider the class `ShoppingSession` in Listing 1. For now, the business logic of this class is not important.

2.1.1. Global Quantification through Wildcards. Let us assume that the programmer wishes to create a performance profile of the e-commerce system. She can easily do so by writing

```
class ShoppingSession {
  ShoppingCart cart; double totalValue;
  void checkOut(Item item, double price, int amount, Customer cus){
    cart.add(item, amount); //fill shopping cart
    cus.charge(price);      //charge customer
    totalValue += price;    //increase total value of session
  } ...
}
aspect Profile {
  Object around(): execution(* *(..)) && !cflow(within(Profile)) {
    long bef = System.currentTimeMillis();
    Object res = proceed();
    long duration = System.currentTimeMillis() - bef;
    log(thisJoinPointStaticPart.getSignature(),duration);
    return res;
  } ...
}
```

Listing 1. Shopping session with profiling aspect.

the simple `Profile` aspect shown in the bottom half of the listing. AspectJ here proves really convenient: the pointcut mechanism provides an easy way to globally quantify over all method executions outside the aspect.

Many programmers may not realize that in this example global quantification only works due to the following two reasons.

—The around advice does not explicitly access any context information from the join points its advises—all context information is accessed through reflection, using **`thisJoinPointStaticPart`**. Because of this, the advice is largely independent of its application context, that is, it can be applied anywhere.
—The around advice uses the return type `Object`, which has a special meaning in AspectJ: it serves as a wildcard, informing the compiler that the advice should apply to join points with just any return type, basically ignoring type checks on the return type. In the example, the profiling aspect therefore applies to executions of `checkOut`, even though it has return type **`void`**, not `Object`.

It is important to highlight that the flexibility provided by the `Object` wildcard return type implies that an advice can return incompatible values that will result in runtime type errors. We come back to typing issues in Section 2.1.3.

2.1.2. Selective Quantification and Fragile Pointcuts. As a matter of fact, not all aspects require such unbounded global quantification. Rather, some aspects are applied to specific program points, closely interacting with the execution context that they advise. In our example, assume a business rule that states that, on his/her birthday, a user of the e-commerce system is given a 5% discount when checking out a product.

Listing 2 shows an implementation of the example where the business rule is defined as an AspectJ aspect `Discount`. The around advice in lines 5–9 applies the discount by reducing the item price to 95% of the original price when proceeding on the user's birthday.

Note how such selective quantification can easily become very brittle with respect to changes in the advised code. Most changes to the signature of the `checkOut` method, such as renaming the method or modifying its parameter declarations, will cause the `Discount` aspect to lose its effect. The root cause of this problem is that the aspect, through its pointcut definition in lines 2–3, references named entities of the base

```
 1 aspect Discount {
 2   pointcut checkingOut(double price, User usr): //User is a supertype of Customer
 3     execution(* ShoppingSession.checkOut(..)) && args(*, price, *, usr);
 4
 5   void around(double price, User usr):
 6     checkingOut(price, usr) {
 7     double factor = usr.hasBirthday()? 0.95 : 1;
 8     proceed(price*factor, usr);
 9   }
10 }
```

Listing 2. Shopping session with discount aspect.

code—here to the `checkOut` method. Of course, after a change to the method signature, regular method calls to the `checkOut` method would also break, but the Java type checker would identify the problem and report errors. On the other hand, implicit invocation through pointcuts will fail silently. The only observable consequence is that the advice does not apply anymore. Some approaches, like pointcut rejuvenation [Khatchadourian et al. 2009], aim at alleviating this problem by reporting changes in aspect matches as programs evolve. But this issue is out of reach for a standard type system.

2.1.3. Unsound Typing of Argument and Return Types. AspectJ has a number of typing issues which have been identified in the literature [Jagadeesan et al. 2006; De Fraine et al. 2008]. First, observe that the advice inside the `Discount` aspect operates on objects of the type `User`, not `Customer` as the advised code does. This is possible because AspectJ allows for variant matching of argument types. The **`args`** pointcut in line 3 binds not only objects of type `User` to `usr` but also objects of any subtype. In general, this is quite convenient for the programmer, as she could therefore bind the same discount advice also to join points that expose users that are not customers.

Although flexible, variant typing of argument types is unsafe. As an example, consider the following advice, bound to the same pointcut, which implements a security feature by replacing any `User` object with an object of the subtype `Guest`.

```
void around(User usr): checkingOut(*, usr) { proceed(new Guest()); }
```

Because `Guest` is not a subtype of `Customer`, the `checkOut` method will fail at runtime with a `ClassCastException` as it requires a `Customer` object.

If we now consider return types, it is easy to see that the treatment of `Object` as a wildcard makes it easy to write an advice that returns values of a type that the advised code is not prepared to handle. For instance, see the following

```
Object around(): call(Customer *(..)) {
    return new Guest();
}
```

The preceding code is type correct in AspectJ, but it returns `Guest` into contexts in which `Customer` objects are expected, causing similar exceptions at runtime.

When the return type of the advice is not `Object`, AspectJ properly ensures that the advice does not return values of incompatible types. For that it checks that the return type of the advice is a subtype of the return type of all join points matched by the associated pointcut. However, this subtyping rule is also unsafe. The problem originates from the fact that the call to **`proceed`** in the advice body is given the same type as the advice itself. For instance, we have what follows.

```
Guest around(): call(User *(..)) {
    Guest ru = proceed(); return ru;    //User may not be a Guest
}
```

The previous code produces a runtime exception, this time in the body of the advice itself: `proceed` is incorrectly assumed to return a value of type `Guest`, whereas it can return any value of type `User`.

2.1.4. Flexibility vs. Type Safety. The aforesaid examples illustrate a trade-off that is inherent to aspect-oriented programming languages with implicit announcement and quantification. Flexible quantification seems to be at odds with type safety. Indeed, avoiding the typing issues given before can be achieved by imposing invariant matching of argument and return types, but that in turn hinders the convenient quantification mechanism that makes AspectJ so popular.

This trade-off is particularly problematic when it comes to defining an interface abstraction that is meant to allow for a proper decoupling of base code and aspects, supporting modular type checking. The design of Join Point Interfaces, and its evolution, is driven by the desire to preserve type safety without abandoning—as much as possible—the flexibility that makes AOP practical.

2.2. Approaches to Aspect Decoupling

We now step back and discuss several approaches to better support decoupling between aspects and base code that have been proposed in the literature. This allows us to set up the context of our proposal more precisely.

2.2.1. Decoupling with Explicit Announcement. In order to obtain better decoupling of aspects, several approaches have been developed that simply avoid implicit announcement, and rely instead on explicit announcement of join points. In these approaches [Rajan and Leavens 2008; Hoffman and Eugster 2007; Eugster and Jayaram 2009; Bodden 2011], base code explicitly denotes expressions or statements that should be exposed as join points. Explicit announcement makes dependencies visible and is therefore more robust; it also simplifies typing. Another advantage of explicit announcement is that it can sometimes be simpler to explicitly denote a region of code to be advised than writing a precise and robust pointcut that exactly identifies that region. Like Steimann et al., we believe that both implicit and explicit announcement are useful, and therefore aim at supporting both mechanisms. This means that the preceding challenges with respect to implicit announcement with quantification need to be addressed.

2.2.2. Decoupling with Implicit Announcement. There is a very large body of work that is concerned with modularity issues raised by the form of implicit invocation with implicit announcement provided by aspect-oriented programming languages like AspectJ, starting with Gudmundson and Kiczales [2001]. In the AOP literature, many proposals have been formulated, some aiming at providing more abstract pointcut languages (e.g., [Gybels and Brichau 2003]), and others—as we do here—introducing some kind of interface between aspects and base code. We now concentrate on the most salient and most related proposals. We discuss further related work in Section 7.

Aspect-Aware Interfaces. Kiczales and Mezini [2005] argue that when facing crosscutting concerns, programmers can regain modular reasoning by using AOP. The particular notion of modular reasoning that is being considered is that of “being able to make decisions about a module while looking only at its implementation, its interface and the interfaces of modules referenced in its implementation or interface”. They introduce an extended notion of interfaces for modules, called aspect-aware interfaces, which can only be determined once the complete system configuration is known. While the argument points at the fact that AOP provides a better modularization of crosscutting concerns than non-AOP approaches, it does not do anything to actually enable modular type checking and separate development. Aspect-aware interfaces are the conceptual backbone of current AspectJ compilers and tools which need to perform checks at weave time.

Open Modules. Aldrich formulated the first approach for recovering modular reasoning through a language-enforced mechanism, Open Modules [Aldrich 2005]. Here, modules are properly encapsulated and protected from being advised from aspects. A module can then open up itself by exposing certain join points, described through pointcuts that are now part of the module's interface. Open Modules give up the flexibility of global quantification to recover local reasoning: aspects rely on pointcuts for which the base code is explicitly responsible. Aldrich formally proves that this allows replacing an advised module with a functionally equivalent one (but with a different implementation) without affecting the aspects that depend on it. Ongkingco et al. [2006] have implemented a variant of Open Modules for AspectJ.

Crosscutting Interfaces. Sullivan et al. [2010] explored a different approach based on a more semantic notion of modular reasoning, directly related to the original work of Parnas [1972]. According to Parnas, a software is modular if it allows for separate development and successfully anticipates future changes. In that sense, modular reasoning is a result of a proper design, which is to a large extent achievable even if the programming language does not provide machine-checked mechanisms that support such decomposition.

In that spirit, XPIs are design rules that aim at establishing a contract between aspects and base code in standard AspectJ. With the XPI approach, aspects generally only define advices but no pointcuts. The pointcuts, in turn, are defined in another aspect representing the XPI. This additional layer of indirection improves the system evolution because the resulting XPI is a separate entity and hence can be agreed upon as a contract between developers. The authors also show how certain parts of such a contract can be checked automatically using static crosscutting or contract-checking advices in the XPI aspect itself.

XPIs are interesting in that they do not sacrifice any of the flexibility of implicit announcement and global quantification as provided by AspectJ. However, without a fully language-enforced mechanism, XPIs cannot provide any strong guarantees towards an error-free integration between separately developed modules. Also, by virtue of being a design approach, XPIs obviously do not address the type soundness issues of AspectJ.

In this work we are interested in introducing a typed abstraction between aspects and base code that enables modular type checking in a safe and flexible manner. In that respect, the closest proposal is Join Point Types (JPTs) [Steimann et al. 2010], which we next discuss in detail.

2.3. Join Point Types

We introduce Join Point Types (JPTs) in detail, to explain the main concepts, their benefits, and—for our purposes of safe modular type checking—their limitations. This understanding is necessary because our proposal for Join Point Interfaces takes JPTs as a starting point. This section focuses on the issues of JPTs that are most relevant to the main argumentation line of this article: quantification, typing, and advice dispatch. For completeness, other issues are discussed in Section 7.

2.3.1. Introduction to JPTs. Listing 3 shows the previous example expressed with JPTs. A JPT is defined as a plain data structure: lines 1–3 define the join point type `CheckingOut`

```
 1 joinpointtype CheckingOut {
 2   double price; Customer cus;
 3 }
 4 class ShoppingSession exhibits CheckingOut {
 5   pointcut CheckingOut: execution(* checkOut(..)) && args(*, price, *, cus);
 6   //remainder of code as before
 7 }
 8 aspect Discount advises CheckingOut {
 9   void around(CheckingOut jp) {
10     double factor = jp.cus.hasBirthday()? 0.95 : 1;
11     jp.price = jp.price * factor;
12     proceed(jp);
13   }
14 }
```

Listing 3. Shopping session example with JPTs.

along with the names and types of context parameters that join points of this type expose. The base class `ShoppingSession` is enhanced to declare that it *exhibits* join points of type `CheckingOut`. In line 5 the class binds the join point type to concrete join points, using a regular AspectJ pointcut.

Through the use of join point types, the aspect is completely liberated from pointcut definitions. Note that in line 9 the aspect refers directly to the join point type, eliminating any reference to advised code elements. This design therefore properly decouples advised code and aspects through a typed abstraction.

2.3.2. Quantification Issues. With JPTs, classes have to explicitly declare that they expose join points of a certain type, through an **`exhibits`** annotation, and then define the corresponding pointcut. This mechanism is called *polymorphic pointcuts*, because each class is responsible for its own exposure behavior, similarly to how each class is responsible for implementing its own methods.

This approach sacrifices the global quantification that AspectJ supports. Aspects that involve broadly crosscutting pointcuts, like profiling in Listing 1, are not easily expressed with JPTs. Instead programmers must expose join points of the given JPT in each class.

The idea of supporting global quantification originates from JPTs [Steimann et al. 2010, Section 3.1]. There, such a mechanism is first considered but then discarded in favor of the mechanism of polymorphic pointcuts because, as the authors argue, global quantification, by its very nature, hinders modular reasoning [Steimann 2013]. Both the language specification [Steimann et al. 2010, Section 4] and its implementation do not support global quantification, nor is it empirically evaluated. The lack of global quantification "seems to be the necessary price for achieving modularity" [Steimann et al. 2010, page 24]. In our view, however, it is unsatisfactory not to be able to accomodate a number of successful applications of AOP like dynamic analyses.

2.3.3. Typing Issues. While JPTs are a typed abstraction to decouple advised code and aspects, the proposed type system is unsound. Notice that the join point type definition in lines 1–3 of Listing 3 specifies neither the return type of these join points, nor the checked exceptions that the execution of these join points may throw.

Because this information is lacking, JPTs must defer checking of some conditions, such as return type compatibility, to weave time or even runtime. The situation, while similar to that of AspectJ, is arguably more problematic because typed interfaces are supposed to enable separate development and modular type checking.

To understand the gravity of the problem, one can make the analogy with interfaces in statically typed languages like Java. Interfaces allow for modular type checking and independent development. However, if interfaces only included type information about method parameter types, such as

```
interface Printer {  print(Document d);  }
```

it would be easy to see that the type system would not be able to maintain soundness due to the lack of information about the return type and checked exception types of method `print`. Both implementors and clients of this interface would have to make informed guesses about those types, resulting in fragile code due to hidden dependencies.

To illustrate the problem with checked exceptions, which we have not elaborated on earlier, consider that some requirements changed on the aspect side. To determine a customer's birthday the aspect now has to submit an SQL query.

```
void around(CheckingOut jp) throws SQLException {
        boolean hasBirthday = SQL.query(/*query omitted*/);
        double factor = hasBirthday ? 0.95 : 1;
        jp.price = jp.price * factor;
        proceed(jp);
}
```

SQL queries can cause checked `SQLExceptions`, and the aspect programmer, being careless, decides to just pass the exception on, by including `SQLException` in its **`throws`** clause. Nothing in the JPT type system prevents the programmer from doing so. Since the join point type definition contains no information about exceptions, a static type checker based on this definition cannot tell which exceptions are allowed or not. As a result, the type error remains latent until system integration time: weaving will fail and report an error if the advice is woven into a context in which `SQLExceptions` are unexpected.

Finally, JPTs also inherit from AspectJ the same unsoundness problems with respect to variant argument and return types, which we discussed in Section 2.1.3 and therefore do not reiterate here.

2.3.4. Problem of Join Point Duplication. Another very useful feature of JPTs is the ability to declare join point subtypes. Such subtypes typically describe more restricted sets of join points than their supertypes, and they may expose additional context information. For instance, consider that we introduce two subtypes of `CheckingOut` from our running example.

```
joinpointtype BuyingBestSeller  extends CheckingOut { String
    bestSellerName; }
joinpointtype BuyingEcoFriendly extends CheckingOut { }
```

The interesting question about such join point subtypes is in what way join points of such types should be dispatched to advices. Within each aspect, the most specific advice defined for the actual type of the join point is executed. Consider the occurrence of a `BuyingBestSeller` join point, where an aspect implements an advice for the `CheckingOut` JPT. The advice is executed as if the join point were of type `CheckingOut`, ignoring the additional `bestSellerName` argument. If an aspect instead implements an advice for both `CheckingOut` and `BuyingBestSeller`, then only the latter will execute, potentially making use of the additional `bestSellerName` argument.

The preceding description tends to suggest that JPT supports *join point polymorphism*, that is, the ability to consider join points at different types. An interesting corner case, though, reveals the fact that JPTs implement join point duplication rather than polymorphism. Consider the situation in which a join point is of two sibling types, for example, `BuyingBestSeller` and `BuyingEcoFriendly`. This can occur when the respective pointcuts associated with those types overlap, that is, match a nondisjoint set of join points. Now consider an aspect that only implements an advice for `CheckingOut`, which sends out a notification email.

The advice dispatch semantics of JPTs results in the *same* advice being executed *twice* for the *same* underlying join point. This is because the semantics of JPT is that of *join point duplication*, rather than polymorphism. In effect, for a given join point, the implementation creates one join point type instance per type assigned to the join point. In the example, it creates one join point of type `BuyingBestSeller` and one join point of type `BuyingEcoFriendly`.

To us, this semantics is confusing. After all, a join point is "a principled point in the dynamic execution of a program" [Kiczales et al. 2001], which conceptually cannot be duplicated. Executing the same advice twice for the same join point (e.g., a single call to method `checkOut`) is surprising.[2] A true polymorphic treatment of join point types would rather allow a single join point to be seen at different types, without inducing duplication of any kind.

3. JOIN POINT INTERFACES

The main contribution of this article is the design, implementation and evaluation of join point interfaces, a novel interface abstraction that combines ideas from previous work together with novel concepts to obtain a language design that is first and foremost (to the best of our knowledge) type sound, but yields a flexibility similar to the one that users are used to from plain AspectJ. Specifically, join point interfaces reuse but refine the ideas of join point types from Steimann et al. [2010], incorporating ideas from the work of Jagadeesan et al. [2006] on generic advice, and innovating in support for global quantification and join point polymorphism. The main contribution of this article is to make all these concepts work together in a sound and seamless fashion, to yield a language design that promises to be of practical use.

3.1. Introduction to JPIs

In the following we describe JPIs from a user viewpoint, discussing how the language design of JPIs addresses the problems previously identified.

3.1.1. Defining Join Point Interfaces. Considering the running example from previous sections, programmers can use the following join point interface definition to decouple the aspect definition from the advised base code.

```
jpi void CheckingOut(double price, Customer cus);
```

In the core language design, join point interfaces are, except for the **`jpi`** keyword, syntactically equivalent to method signatures. This is for a good reason: methods are designed to be modular units of code that can be type checked in a modular way, and we wish join point interfaces to enjoy this property. Notably, join point interfaces include both return and checked exception types, which pointcuts and join point types are missing.

3.1.2. Implicit Announcement with Pointcuts. Programmers can raise join points of a certain JPI through either implicit or explicit announcement. Implicit announcement uses

[2]Steimann et al. actually acknowledge this problem: "This situation may appear somewhat awkward. [...] We leave this debate for future exploration." [Steimann et al. 2010, page 16].

```
 1 jpi double CheckingOutR(double price, Customer cus); //include double return type
 2 
 3 class ShoppingSession { ...
 4     void checkOut(final Item item, double price, final int amount, final Customer cus) {
 5        totalValue = exhibit CheckingOutR(double alteredPrice) {
 6                        cart.add(item, amount);
 7                        cus.charge(alteredPrice);
 8                        return totalValue + alteredPrice;
 9                    } (price);
10     }
11 }
```

Listing 4. Example from Figure 2 with closure join points.

pointcuts, just like in AspectJ or JPT. For instance, the class `ShoppingSession` can exhibit the appropriate `CheckingOut` join points as follows.

```
class ShoppingSession {
    exhibits void CheckingOut(double price, Customer c):
        execution(∗ checkOut(..)) && args(∗, price, ∗, c);
    ...
}
```

In this piece of code, the programmer raises join points implicitly, through an **`exhibits`** clause that defines an associated pointcut. Within the JPI language, a pointcut attached to an **`exhibits`** clause only matches join points that originate from a code fragment lexically contained within the declaring class. Just like in JPT, **`exhibits`** clauses do not match join points in subclasses. This avoids an overly complex semantics, as previously observed by Steimann et al. [2010]. There are two important differences compared to JPTs: quantification includes inner classes (we come back to this in Section 3.2), and aspects themselves can exhibit join points. (In the following, when we talk about a classes or base-code that exhibits join points, we therefore also include aspects.)

3.1.3. Explicit Announcement with Closure Join Points. JPIs support explicit announcement through a language construct called Closure Join Points [Bodden 2011]. Closure join points are explicit join points that resemble labeled, instantly applied closures, that is, anonymous functions with access to their declaring lexical scope. Closure join points were first proposed independently of JPIs; their syntax and semantics solves several subtle semantic issues that arise with the explicit announcement in the JPT implementation (discussed elsewhere [Bodden 2011]). Join point interfaces seamlessly integrate with closure join points.

Listing 4 shows the shopping-session example from Listing 2 adapted to use explicit announcement. Instead of using a pointcut in an **`exhibits`** clause, the programmer exposes a sequence of statements (lines 6–8) using a closure join point of the JPI `CheckingOutR`. This JPI is identical to `CheckingOut`, except for the fact that it declares a return type **`double`**. Note that an advice is oblivious to whether a join point is announced implicitly or explicitly; only its type—as specified by one or more JPIs—matters. (Note that, similar to how Java objects can implement multiple interfaces, join points can be typed with several JPIs. We will get back to this in Section 3.4.)

3.1.4. Advising JPIs. The following piece of code shows how programmers can advise join points of a given join point interface.

```
aspect Discount {
    void around CheckingOut(double price, Customer c) {
        double factor = c.hasBirthday()? 0.95 : 1;
        proceed(price*factor, c);
    }
}
```

As with JPTs, advices for JPIs only refer to join point interfaces, not to pointcuts or base-code elements. The main difference with JPTs is that the advice signature matches the JPI, and hence receives context information exposed at join points via formal parameters, as in AspectJ. In our design, join points are not first class.

3.2. Flexible Quantification

While join point interfaces generally follow the same approach as JPTs with respect to decoupling, we aim at designing a solution for widely quantifying aspects like dynamic analyses. This implies some trade-offs with respect to modularity and separate development, which we discuss shortly. The combination of local and global quantification that JPIs provide is in itself a novel answer to the quantification design question.

3.2.1. Global Quantification. Having to define pointcut expressions within every class that requires advising does not scale very well when trying to implement dynamic analysis aspects such as tracing or runtime checking, which require wide global quantification. We allow programmers to specify a *global pointcut*, together with the definition of a JPI. We call such a globally quantifying JPI a global JPI.

For the profiling aspect from Listing 1, an appropriate global JPI definition could look as follows.

```
global jpi Object AllExec(): execution(* *(..));
```

Note that this uncontrolled use of global quantification reintroduces implicit dependencies between the global JPI and the base code. Indeed, the base code does not directly refer to the JPI, hence modular reasoning is hindered.

The situation is still better than in plain AspectJ, however, as the base-code programmer needs to be aware of the JPIs only, independently of the specific aspects that intend to advise these JPIs. So aspects and base code can be separately developed. In a team development scenario, programmers could even agree upon defining global JPIs in a single file, making the implicit dependencies entailed by global quantification more easily tractable. Highlighting support by the development environment such as in the AspectJ Development Tools [Clement et al. 2003] could aid base-code programmers further, without requiring access to aspect definitions. Nevertheles, JPIs do offer users additional control over global quantification if desired.

3.2.2. Controlled Global Quantification. Global quantification can be controlled in a fine-grained manner. For instance, any class or aspect may seal itself to prevent global quantification from applying to its own definition.

```
sealed class Secret { ... }
```

Join points raised lexically inside the `Secret` class are hidden to global pointcuts. A sealed type can opt to exhibit join points explicitly if desired; sealing just impacts global pointcut definitions. For instance, the following class is immune to the `AllExec` global pointcut, but exposes certain `Caching` join points.

```
sealed class Secret {
    exhibits Object Caching() : ...
}
```

Further, programmers can opt to refine global pointcuts that they are aware of. When doing so, the declared global pointcut is a available as a named pointcut **global** inside an **exhibits** clause.

```
sealed class Secret {
    exhibits Object AllExec() : global() && !call(* secret(..))
}
```

Here, class `Secret` allows all its method execution join points to be matched by the `AllExec` global pointcut, but selectively hides the executions of a `secret` method. Sealing a class eliminates implicit dependencies, because it can only be advised through (possibly global) JPIs that are explicitly referenced in its definition.

3.2.3. Open and Sealed Classes. So far, we have considered that, by default, classes are open to global quantification, and that classes that want to prevent global advising need to be explicitly declared as **sealed**. The dual approach is also appealing: by default, all classes are sealed, and classes that approve global advising need to be explicitly declared as **open**. The choice of a good default reflects a particular inclination towards either favoring backward compatibility with the way programmers are used to program in current aspect languages like AspectJ, or favoring instead modularity in a more conservative manner.

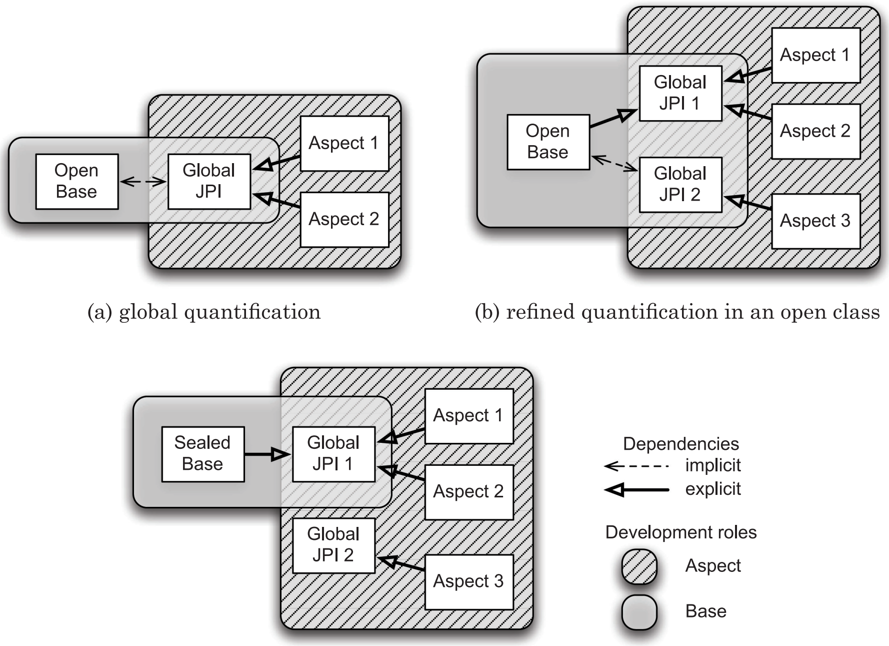
Because we believe that there is no absolute response to this language design question, we actually support both default semantics in our implementation.[3] This means that it is possible to declare a class or aspect as **open**.

```
open class NothingToHide { ... }
```

If the language default is set to sealed-by-default, then the previous declaration allows the class `NothingToHide` to be exposed to global quantification. (Otherwise it is just redundant.)

Figure 2 summarizes the different scenarios in presence of global quantification. First, Figure 2(a) shows how unrestricted global quantification over an open class implies fragile dependencies between the base code and the global JPI. Note that this situation is still “better” than the AspectJ situation (Figure 1(a)), because the aspects themselves do not have implicit dependencies. Figure 2(b) depicts the situation in which an open class refines the global JPI 1. The implicit dependency to JPI 1 is replaced by an explicit dependency, but an implicit dependency to the other global JPIs remains. Finally, Figure 2(c) shows how a sealed class that chooses to refine a global pointcut enjoys the same benefits as with nonglobal JPIs (Figure 1(b)): no implicit dependencies are left. In fact, in this case the only difference to the the scenario with a nonglobal JPI is that the global JPI provides a default pointcut **global** whose use can ease the definition of the **exhibits** clause.

[3]At present, the default is that of explicit sealing, but this can be changed through a compiler flag. Supporting more fine-grained specification of the semantics of unannotated classes, for instance at the package (JAR) level, is left for future work.



(a) global quantification (b) refined quantification in an open class

(c) refined quantification in a sealed class

Fig. 2. Quantification variants and impact on dependencies.

3.2.4. On Local Quantification. Join Point Types and Open Modules have different takes on local quantification. With Open Modules, classes themselves do not declare their exposed join points; it is the task of the module. The argument is that the maintainer of the module is in charge of all the classes inside the module, and therefore has sufficient knowledge to maintain classes in sync with the pointcuts in the module interface. With JPTs, quantification is class local: each class is responsible for what it exhibits. This class locality is very strict: even *nested* classes are not affected by the exhibited pointcuts of their enclosing classes.

In this work, we do not extend Java with a new notion of modules (this is left for future work), but we do support nested classes in the sense that the exhibited pointcut of a class matches join points in nested classes as well. This means that we can use class nesting as a structuring module mechanism, and obtain module-local quantification, as in Open Modules. Of course, this approach to modules is certainly not as well supported in Java as it would be in a language like Newspeak [Bracha et al. 2010], in which modules are objects, supported by a very flexible virtual class system. We believe that module-local quantification is a good intermediate situation between strict class-local quantification and global quantification. If Java ever obtained a standardized language-level module system, then **`exhibits`** clauses would be part of module declarations.

3.3. Modular Type Checking

We now turn to the issue of modular type checking, and how to reconcile it with flexibility.

3.3.1. Invariance. To avoid the type unsoundness caused by type variance in argument and return types previously explained for AspectJ (Section 2.1.3), the JPI type system enforces that an advice for a JPI such as `CheckingOut` carries a signature whose argument and return types coincide exactly with those of the JPI declaration. In other words, we enforce type invariance. This avoids the type unsoundness that users may experience with AspectJ.

```
void around CheckingOut(double price, Customer cus) {
    proceed(price, new Guest()); //type error
}
```

In this example, due to type invariance, the injection of objects of incorrect types such as `Guest` given before is no longer possible. (Remember that `Guest` is not a subtype of `Customer`.) Similarly, the return type of the advice cannot be a subtype of the return type of the JPI, in order to avoid runtime type errors in the advice body. Additionally, the `Object` return type is not a wildcard; it really means `Object`.

3.3.2. Checked Exceptions. Join point interface signatures must be complete with respect to the checked exceptions they declare. The preceding JPI definition declares no exception, which gives both the aspect and the base code the guarantee that the respective other side of the interface cannot throw any checked exceptions at join points of this type. If needed, the programmer can declare checked exceptions in the join point interface.

```
jpi void CheckingOut(double price, Customer cus) throws SQLException;
```

In this example, both the advised join point and the handling advice are allowed to throw `SQLExceptions`, but must also take care to handle (or forward) the exceptions appropriately. For instance, here are two possible advice definitions.

```
void around CheckingOut(double price, Customer cus) {
    try{ proceed(price, cus); } catch(SQLException e) { ... }
}
void around CheckingOut(double price, Customer cus) throws SQLException {
    proceed(price, cus);
}
```

The first advice handles `SQLExceptions`, while the second advice simply forwards them. Both are valid because the base code has to be prepared anyway to handle such exceptions. This is because the base code cannot know if and how the join point is going to be advised. The type checker therefore conservatively assumes, as defined in the JPI, that exceptions of type `SQLException` may be thrown in any case. It seems that a way to distinguish potentially handled from definitely handled exceptions could prove useful, but we leave this question to future work.

3.3.3. Regaining Flexibility through Generic Types. Especially in combination with global quantification, the invariant typing of JPIs can be too rigid, preventing certain join points from being advised. Recall the global JPI definition from Section 3.2.

```
global jpi Object AllExec(): execution(* *(..));
```

Due to invariant typing, this definition does not achieve what the user actually intends, as it only selects join points whose return type is exactly `Object`, and which also do not throw any checked exceptions.[4]

[4]In practice this would actually cause error messages because the pointcut matches join points that are not type compatible with the JPI signature. This is similar to the matching errors raised by the AspectJ compiler when the return type of an advice does not coincide with the type of a matched join point.

```
jpi void Renting(double price, int amount, Customer c) extends CheckingOut(price, c);

aspect Discount {
  void around CheckingOut(double price, Customer cus) { /* as before */ }
  void around Renting(double price, int amt, Customer cus) {
    double factor = (amt > 5) ? 0.85 : 1;
    proceed(price*factor, amt, cus);
  }
}
```

Listing 5. Advice overriding.

To regain flexibility while preserving type safety, we resort to parametric polymorphism and support generic JPIs. This is directly inspired by work on generic advice by Jagadeesan et al. [2006], which was developed further by De Fraine et al. [2008]. A proper generic definition of the `AllExec` JPI is as follows.

```
<R,E> global jpi R AllExec() throws E: execution(* *(..));
```

In this definition, we explicitly abstract from the return type and any checked-exception type that the method may declare. (Note that the type variable `E` is special in the sense that it can represent any number of declared exceptions. We explain the semantics later in this article.) As in Java, type variables can have an upper bound; this allows code advising a generic JPI to make stronger assumptions about the concrete types at the respective join points.

```
<U extends User> jpi void CheckingOut(double price, U user);

<U extends User> void around CheckingOut(double price, U user) {
    log(user.getName()); //succeeds (assuming getName() is defined on
        User)
    proceed(price, new User()); //fails: User is not a subtype of U
}
```

Note that the advice is allowed to proceed only with objects of type `U`. This is because `U` denotes some concrete subtype of `User`, not `User` itself, and therefore `User` is not a subtype of `U`. In practice, the only valid values to proceed with are thus either `user` itself, the result of `user.clone()` (which, due to `clone()` having return type `Object`, the programmer would need to cast to `U`), or a reflectively instantiated peer object of `user`. The same restriction applies to values returned by **proceed** and by the advice. Section 4.2.4 gives further information on the semantics of type variables used for checked exceptions.

3.4. Polymorphic Join Points and Advice Dispatch

A join point interface denotes the type of a join point. A join point can be of different types: for instance, if both the `CheckingOut` and `AllExecs` JPIs are used, then the join point that corresponds to the execution of the `checkOut` method has both types. JPIs implement an advice dispatch semantics that guarantees that every advice executes at most once for each join point, thereby avoiding the multiple execution problem of JPTs (Section 2.3.4). Section 4.3 describes this semantics in details.

Also, a JPI can extend another JPI. Consider two subtypes of `CheckingOut`, `Buying` and `Renting`, and the following business rule: the customer gets a 15% discount when *renting* at least five products of the same kind; this promotion is not compatible with the birthday discount. Listing 5 shows an implementation of this additional rule using

Table I. Summary of Features and their Purpose

Feature	Purpose
Join point interfaces	Decoupling aspects and base code
JPIs as full method signatures	Complete specification for modular checking
Class-local pointcut matching	Limit pointcut fragility
Closure join points	Support explicit announcement
Controlled global pointcuts	Support wide quantification
Invariant typing of argument, return, and exception types Invariant pointcuts	Preserve type soundness
Parametric polymorphism	Enhance the flexibility of the type system
Join point polymorphism	Support expressive modeling and handling of events; no duplication of join points

subtyping on JPIs. First, we declare the JPI `Renting` as extending `CheckingOut`. The semantics of this subtyping relationship implies that any join point of type `Renting` is also a join point of type `CheckingOut`. The **`extends`** clause defines how arguments are passed to superjoin points, similar to primary constructors in Scala [Odersky et al. 2008]. In the example, the first and third arguments of `Renting` join points become the first, respectively second, argument when this join point is seen through the `CheckingOut` interface.

This effect can be seen in the aspect `Discount`, which now declares two advices. The first one is the same as in the previous example. In general, it applies to all `CheckingOut` join points, and if this advice was the only advice in the aspect, then it would indeed execute also for join points of type `Renting`. In the example, however, the aspect defines a second advice specifically for `Renting`. In this case, an overriding semantics applies: the more specific advice overrides the first advice for all join points that are of type `Renting`. As a result, the first advice only executes for join points of other subtypes of `CheckingOut`, that is, for `CheckingOut` itself, or for `Buying`.

3.5. Summary

Table I summarizes the main features of Join Point Interfaces and the purposes they fulfill. When designing the JPI mechanism, and faced with a tension between type safety and flexibility, we always opted in favor of type safety first. We then tried to recover flexibility by introducing new mechanisms, without sacrificing safety. For instance, the first version of JPIs [Inostroza et al. 2011] only had invariant typing, which is safe but sometimes too rigid. Similarly, the first incarnation of JPIs did not have any support for global quantification. Parametric polymorphism and controlled global quantification were introduced to recover flexibility. As a result, we propose an integrated language design that provides strong typing guarantees, supports separate development, and can accommodate a smooth transition path from standard AOP practice.

4. SYNTAX AND SEMANTICS OF JPIs

We begin this section by describing the syntax of JPIs, designed as an extension to AspectJ (Section 4.1). Join point interfaces aid the separate development of aspect-oriented programs by precisely mediating the dependencies between aspects and base code. The most fundamental contribution of JPIs therefore lies in the static type system that supports modular type checking: we informally describe it in Section 4.2. Our

TypeDecl ::= . . . | *JPITypeDecl*

JPITypeDecl ::= [“`global`”] “`jpi`” [“<” *TypeParamList* “>”] *JPISig* [*JPIExt*] [“`:`” *PointcutExpr*] “`;`”

JPISig ::= *Type ID* “`(`” [*ParamList*] “`)`” [*ThrowsList*]

JPIExt ::= “`extends`” *Name* “`(`” [*ArgList*] “`)`”

ClassMember ::= . . . | *ExhibitDecl*

AspectMember ::= . . . | *ExhibitDecl*

ExhibitDecl ::= [“<” *TypeParamList* “>”] “`exhibits`” *Type Name* “`(`” [*ParamList*] “`)`”
[*ThrowsList*] [“`:`” *PointcutExpr*] “`;`”

AdviceDecl ::= . . . | [“<” *TypeParamList* “>”] *JPIAdviceDecl*

JPIAdviceDecl ::= [*Modifiers*] [*Type*] *AdviceKind ID* “`(`” [*ParamList*] “`)`” [*ThrowsList*] *Block*

Expr ::= . . . | *CJP*

StmtExpr ::= . . . | *CJP*

CJP ::= “`exhibit`” *Name* “`(`” [*ParamList*] “`)`” *Block* “`(`” [*ArgList*] “`)`” | “`exhibit`” *Name Block*

Modifier ::= . . . | “`open`” | “`sealed`”

Fig. 3. Syntactic extension for Join Point Interfaces (AspectJ syntax is shown in gray).

proposal of JPIs also innovates over previous work in the way it supports join point polymorphism. The dynamic semantics of JPIs is described in Section 4.3.

4.1. Syntax

Figure 3 presents our syntactic extension to AspectJ to support join point interfaces and closure join points. Type declarations, which normally include classes, interfaces, and aspects, are extended with a new category for JPI declarations. A **`jpi`** declaration specifies the full signature of a join point interface: the return type at the join points, the name of the join point interface, its arguments, and optionally, the checked exception types that may be thrown. A join point interface declaration can also specify a superinterface, using **`extends`**. In that case the name of the extended JPI is given, and the arguments of the superinterface are bound to the arguments of the declared JPI. Further, it can specify a global pointcut expression.

Classes and aspects can have a new kind of member declaration, for specifying the join point interfaces that are exhibited. An **`exhibits`** declaration specifies a join point interface signature and the associated pointcut expression that denotes the exhibited join points.

Further, our extension allows advices to be bound to JPIs. Opposed to normal advices, instead of directly referring to a pointcut expression, such advices instead refer to a join point interface. The information about return type, argument types, and checked-exception types that the JPI specifies becomes part of the advice signature.

Closure join points can be used in any place in which an expression can be used. A closure join point comprises the keyword **`exhibit`**, an identifier (the name of the JPI used to type the exposed join point), and a block, plus optionally a list of formal and actual arguments. If those lists are omitted, this is equivalent to specifying empty lists. The block effectively defines a lambda expression, while the formal parameter list defines its λ-bound variables. Opposed to a regular lambda expression, however, a closure join point must be followed by an actual parameter list: the closure is always immediately called; it is *not* a first-class object.

Generic types can be used within JPI declarations, advice declarations, and **`exhibits`** clauses. Closure join points do not need type parameters, as they are actual instances of join points, binding to the type parameters of the corresponding join point interface.

The language extension is backwards compatible with AspectJ, that is, any valid AspectJ program is valid in this language, and the semantics of the program is unchanged. This allows for a gradual migration of AspectJ programs to join point interfaces. It would be simple, though, to include a "strict" mode that instructs the parser to disallow AspectJ's regular pointcut and advice definitions.

4.2. Static Semantics

We next describe the JPI type system, a key contribution of this work. The type system supports modular type checking of both aspects and classes; only knowledge of shared JPI declarations is required to type check either side. This is similar to type checking a Java class based on the interfaces it references.

We first discuss how JPIs are used to type-check aspects. Then we turn to type-checking base code that exhibits certain JPIs. Finally, we discuss type-checking JPIs themselves, in particular considering JPI inheritance.

4.2.1. Type-Checking Aspects. An aspect is type checked just like an AspectJ aspect, save for its advices. There are two facets of type checking an advice: checking its signature and checking its body. Type checking the signature of an advice is simple, but requires care. Each advice declares an advised JPI in its signature. The argument types of the advice must be exactly the same as the ones declared in the JPI. For around advice, the return types must also exactly coincide. As explained in Section 3.1.4, the crucial requirement here is that the typing of all types in the signature is *invariant*.[5]

Checked exceptions require care. Calls to **proceed** are assumed to potentially throw any types of checked exceptions that the JPI, which the surrounding around advice is bound to, declares. An advice must not declare any additional exception, as this could lead to uncaught checked exceptions on the side of the base code. It can, however, declare fewer exceptions if the omitted exceptions are handled within the advice, or declare additional exceptions (e.g., `EOFException`) provided they are subtypes of expected exceptions (e.g., `IOException`) that are also declared in the JPI.

The exceptions that are declared in the advice interface are typed invariantly with respect to the JPI. This strict invariance requirement for exceptions is required for the same reason as for return and argument types. Let us denote T_S the checked exception thrown at a join point shadow[6], T_I the exception type declared in the JPI, and T_A the type of checked exception thrown by the advice. If $T_S < T_I$[7], this means that the context of the shadow is not prepared to handle T_A if $T_S < T_A <: T_I$. Conversely, if $T_S > T_I$, the advice is not prepared to handle T_S when invoking **proceed**. Therefore $T_S = T_I$ by necessity.

Type checking the advice body is similar to type checking a method body, with the additional constraint of considering calls to **proceed**. A join point interface is identical to a method signature (except for the **extends** clause used for join point subtyping). In fact, a JPI specifies the signature of **proceed** within the advice body, thereby abstracting away from the specific join points that may be advised. This is a fundamental asset of JPIs, and the key reason why we believe that interfaces for AOP ought to be represented as method signatures, including return and exceptions types. JPIs *fully* specify the behavior of advised join points as far as the type system is concerned, thereby allowing safe and modular static checking of advice.

[5]In theory, since before/after advice cannot replace any values in the context in which they execute, one could loosen their typing restrictions on argument types. We were unable to find an advantage of doing so, however, which is why we just apply the same restrictive typing rules to before and after advice as well.
[6]A join point shadow is the source expression whose evaluation produces a given join point [Masuhara et al. 2003; Hilsdale and Hugunin 2004].
[7]We use $<:$ for subtyping, and $<$ for strict (i.e., nonreflexive) subtyping.

4.2.2. Type-Checking Base Code. On the other side of the contract is base code, which can exhibit join points. The base code must also obey the contract specified by join point interfaces. Part of this contract has to be fulfilled by the pointcut associated with the **exhibits** clause used for implicit announcement: the pointcut has to bind all the arguments in the signature, using pointcut designators such as **this**, **target**, and **args**. To comply with the invariant semantics, those pointcuts must match invariantly, that is, a pointcut such as `this(A)` must only match join points with declared type `A`. As we will explain in Section 5, this semantics differs from the one of AspectJ.

Because pointcuts do not account for return and exception types, the type system checks these types at each join point shadow matched by the pointcut associated with the **exhibits** declaration. More precisely, the pointcut is matched against all join point shadows in the lexical scope of the declaring class. Whenever the pointcut matches a join point shadow, the type system checks that the return type of this shadow and the JPI coincide. If the shadow has a different return type, the type checker raises an error message stating that the selected join point shadow is incompatible with the JPI in question. Similarly, the type system validates that the declared exceptions of the shadow are the same as those of the JPIs; if they are not, the type checker raises an error. Here again, type compatibility is invariant. Note that those checks, although they require pointcut matching, do not require access to any aspect code.

One may wonder why we raise an error message when pointcuts select incompatible join point shadows and not simply restrict the matching process to exclude incompatible shadows instead, taking into account the return and exception types of the JPI. The reason is to follow as closely as possible the way AspectJ deals with such situations already. In the following example, AspectJ raises an error because the **execution** pointcut selects a join point (in line 2), whose return type **int** is incompatible with the return type **void** of the around advice.

```
1 void around(): execution(int foo()) { ... }
2 int foo () { return 0; }
```

Type checks at join point shadows are the fundamental contribution of JPIs from the point of view of type-checking base code. All previous approaches, particularly JPTs [Steimann et al. 2010], are not able to perform these checks modularly, simply because the specification of return and exception types are not part of the proposed interfaces. These approaches have to defer type checks to weave time, or even worse, to runtime. With JPIs, conformance can be checked statically and modularly, prior to weaving.

Type-Checking Closure Join Points. Type-checking closure join points follows the Java static semantics for methods defined within inner classes. Code within a closure join point has access to its parameters, to fields from the declaring class, and to local variables declared as **final**. Access to non-**final** local variables is forbidden because, by exposing a block of code to aspects in the form of a closure join point, aspects may choose to execute the closure in a different thread, or may choose to not execute the closure at all. In this case, write access to local variables may cause data races or undefined values on those variables [Bodden 2011].

Other than that, we type-check closure join points according to the same rules as implicitly announced join points. To keep the code as concise as possible, closure join points do not define return and exception types; instead those are inferred from the join point interface declaration they reference. Because of this, those parts of the closure join point's signature are automatically invariant with respect to the referenced join point interface. Arguments are deliberately not inferred. This is because we want to allow users to give arguments within the closure join point their own local name, independent of what was specified in the JPI declaration and independent of the context in which the closure join point is declared. This is in line with method definitions and lexical scoping in Java. The independence from the closure join point's declaring context is important to distinguish the values of such variables that are declared in this context from the potentially altered values that an advice may pass as the closure join point's arguments. For instance, in Listing 4 the value of `alteredPrice` may differ from the one of `price`. For the types of those arguments, we check that the declaration in the closure join point's header obeys an invariant typing semantics. Further, we check that the body of a closure join point only throws checked exceptions of types that the referenced JPI declares, and conversely that the declaring context of the closure join point is prepared to handle (or declares to forward) all checked exceptions of these types.

4.2.3. Type-Checking JPIs. Join point interfaces support a subtyping relationship. We employ type checking at the level of JPIs to ensure that subtyping relationships do not break type soundness. As we exemplified in Section 2, any JPI can declare to extend another JPI. During this process, the sub-JPI can add context parameters; JPIs thus support *breadth subtyping*. On the other hand, all arguments that do coincide have to be of the exact same type as the respective arguments in the JPI supertype; JPIs disallow *depth subtyping*. This is due to the same reasons for which we use invariant argument typing in all other situations. For example, the following code would raise a type error, even if `B` were a subtype of `A`.

```
jpi void Base(A a);
jpi void Sub(B b) extends Base(b);  //type error; no variance allowed
```

In addition, JPI subtypes must declare the same return type as their super-JPI, and must declare the same exceptions to be thrown.

4.2.4. Parametric Polymorphism through Generic Types. In general, the semantics of type checking in the presence of parametric polymorphism is exactly the same as that of Java with generics, as defined in Featherweight Generic Java [Igarashi et al. 2001]. If a JPI uses type variables, then any referencing **`exhibits`** clauses and advices must use generic type variables at the same signature positions, with identical type bounds (if present), and vice versa. The names of type variables are scoped locally, that is, the following is allowed.

```
<A> jpi void JP(A x);
<B> before JP(B x) { ... } //is ok even though B!=A
```

When checking base code, a type at a shadow matches a type variable in the JPI if it is a subtype of the bound of that variable. As explained before, this is where flexibility is gained, allowing generic JPIs to be practical in the face of wide crosscutting. Note that soundness is preserved, through parametricity [Reynolds 1983]: advices cannot assign any values to variables of a parametric type (modulo the usage of `clone` or reflection).

We only support upper bounds for type variables (declared with **`extends`**), since they are enough to handle the needs for polymorphism that we have observed in practice through the case study. We leave to future work the exploration of the other features found in object-oriented languages with parametric polymorphism (lower bounds, wildcards, type constraints as in Scala [Odersky et al. 2008], etc.).

As we already showed in Section 3.3.3, programmers can choose to use parameterized types for checked exceptions just as for any other type declaration. An interesting question is how bounds impact type checking for checked exceptions. Imagine the following example.

```
<S extends SQLException, ANY> jpi void J() throws IOException, S, ANY;

<S extends SQLException, ANY> void around J() throws IOException, S, ANY {
    try { proceed(); }
    catch(S s) {            //s is seen as SQLException
        s.getSQLState();
        throw s;
    } catch(IOException i) {
        throw i;
    } catch(ANY a) {        //a is seen as Throwable
        a.getStackTrace();
        throw a;
    }
}
```

This example is type-correct. The advice calls **proceed** and then reacts to a range of possible thrown exceptions, eventually rethrowing them. Rethrowing exceptions is always sound, including for variables of a generic type. Note that the advice may safely throw a **new** IOException(). This is safe, as our type system ensures that the JPI will only be bound to join points that declare to throw IOException. The advice may not, however, throw any other types of exceptions. This includes SQLException, as SQLException is not a subtype of S or ANY. The type bounds are thus helpful only for accessing the members of the exception object at a certain type. For instance, getSQLState() is only declared on type SQLException. For type bounds on type variables used for checked exceptions, we enforce that the bounds are a subtype of Throwable.

4.2.5. Global Join Point Interfaces. The type system enforces that the **global**() pointcut designator is only (and always) used in **exhibits** clauses that bind to a JPI that is itself declared as **global**. The signature of the **global** pointcut is defined through the signature of the JPI that defines this pointcut.

When refining a global pointcut with an **exhibits** clause, programmers may provide no pointcut at all. In that case, the class or aspect is sealed against the global pointcut of the specific JPI.

The modifiers **open** and **sealed** must only occur in front of class and aspect declarations, including inner classes. If an inner class declares no such modifier then its sealing status implicitly defaults to that of its enclosing class. (Note that we could even offer sealing on the method level, a feature we plan to investigate in the future.)

4.3. Dynamic Semantics

The dynamic semantics of JPIs differs slightly from that of a traditional aspect language. Briefly, the traditional model is as follows [Wand et al. 2004]: all aspects (pointcuts and associated advices) are present in a global environment; at each evaluation step, a join point representation is built and passed to all defined aspects; more precisely, the pointcuts of an aspect are given the join point in order to determine if the associated advices should be executed or not.

With JPIs, aspects do not have pointcuts. They advise JPIs, and base code defines the join points that are of these types, either using pointcuts (implicit announcement) or using closure join points (explicit announcement). The global environment contains aspects with their advices. With implicit announcement, conceptually, a join point representation is passed only to the pointcuts defined in the current class (and outer classes, if present), at each evaluation step. If a pointcut matches, then the join point is tagged with the corresponding JPIs. Then, the advices that advise one of the tagged JPIs are executed.

In the presence of join point polymorphism and inheritance among JPIs, it is interesting to ask which advice is executed. We write A_T to denote an advice of aspect A that advises JPI T; we write jp^T to denote a join point jp tagged with JPI T. The semantics of advice dispatch is directly inspired by the semantics of message dispatch in multiple dispatch languages like CLOS [Paepcke 1993] and MultiJava [Clifton et al. 2000]. Indeed, an aspect with its multiple advices (each declared to advise a specific JPI) can be seen as a generic function with its multiple methods. Once a join point jp is tagged with interfaces $T_1, \ldots, T_n$ we select, for each aspect A, all applicable advices. An advice A_S is *applicable* to $jp^{T_1,\ldots,T_n}$ if there exists an i such that $T_i <: S$. In order to support overriding, among all applicable advices $A_{S_1}, \ldots, A_{S_k}$, we invoke only the *most specific* ones, defined as the A_{S_j} such that for all i, either $S_j <: S_i$ or $S_i \not<: S_j$.

Aspect-oriented programming, like any publish-subscribe mechanism, inherently supports *multiple reactions* to a single event. This differs from multiple dispatch, which requires exactly one method to execute. The difference manifests in two ways in the semantics. First, if there are no applicable advices, then nothing happens; no advice executes. In contrast, a multiple-dispatch language throws a *message-not-understood* error if no applicable method can be found. Also, message dispatch requires that there is a *unique* most specific applicable method, otherwise an *ambiguity* error is raised[8]. In our case, we execute all the most specific applicable advices, in the precedence order imposed by regular AspectJ when multiple advices of a same aspect match the same join point [Laddad 2003].

In the preceding, we have overlooked one specificity of AspectJ and most aspect languages: the fact that advices can be of different *kinds*—before, after, or around. The advice overriding scheme we described is kind specific: an advice may override another advice only if it is of the same kind. (In Section 6, we will show cases where this is useful.) For instance, consider an aspect that defines two advices A_{T_1} and A_{T_2}, with $T_2 <: T_1$. If one is a before advice and the other is an after advice, both are executed upon occurrence of a join point jp^{T_2}. Conversely, if both are around advices, only the most specific (A_{T_2}) executes, as explained earlier.

In practice, we found that advice overriding is not always desirable (see Section 6.3.3). We support the possibility to declare an advice as **`final`** to prevent overriding. A **`final`** advice always executes if applicable, regardless of whether there exists a more specific applicable advice; in such a case, both execute, in the order defined by the standard AspectJ composition rules.

A fundamental asset of the dispatch semantics presented here is that it gives the guarantee that a given advice executes at most once for any given join point. This is in stark contrast with the semantics of join point types [Steimann et al. 2010], where the same advice can—in our view surprisingly—be executed several times for the same join point, as discussed in Section 2.3.4.

5. IMPLEMENTATION OF JPIs

With this article we provide a full implementation of join point interfaces as an extension to the AspectBench Compiler (abc) [Avgustinov et al. 2005]. The implementation is maintained within abc's own code base.[9] The main goal of JPIs is to facilitate modular type checking and therefore separate development. A secondary goal would be to

[8]In a statically typed language like MultiJava, both cases can be ruled out by the type system.
[9]abc can be downloaded at: `http://aspectbench.org/`.

also allow for separate compilation. Unfortunately, abc was not designed to support separate compilation. The implementation we provide is therefore only a best-effort solution that, to the best of our knowledge, fulfils the primary goal of sound and modular type checking. Given an appropriate compiler implementation, separate compilation can be achieved. We are exploring the possibility to integrate JPIs in the main AspectJ implementation (`eclipse.org/aspectj`), which does support separate compilation.

With this in mind, in the following we first describe an abstract compilation scheme that could provide separate compilation with JPIs. Section 5.2 describes the current abc implementation.

5.1. Separate Compilation with Join Point Interfaces

As described in Section 4, type checking an aspect requires only the code of the aspect itself and the code of all referenced JPIs. Similarly for the base code, type checking requires access to the base code itself, and to all referenced JPIs. If the application uses global JPIs, then the base code requires access to these global JPIs as well, even though they are not referenced explicitly in the base code (recall Figure 2). Because this scenario still allows base code to be independent from actual aspect definitions, we conjecture that this requirement should pose no problem in practice. Global pointcut definitions are part of the global JPI definitions, and are therefore shared as well.

When a code module containing classes (or aspects, since aspects can exhibit join points too) is compiled with a set of JPIs, then the module is first type checked. This involves matching all pointcuts from **`exhibits`** declarations (global pointcut definitions are hereby inlined) against the base code, to determine whether all matched shadows comply with the declared JPIs. Afterwards, aspects are turned into plain Java classes. At this point, the compiler persists metadata information about JPI declarations and pointcuts in class-file attributes. (This is also the approach the Eclipse AspectJ compiler uses for regular AspectJ.) This information would include the information of which JPIs were considered to type check the code.

Later, the programmer would then ask the compiler to weave the individual modules into a ready-to-be-run application. At this point, the compiler would use the metadata information to check that the set of all JPIs is consistent for all modules. If so, then this guarantees that the composition is type safe. Second, the compiler would use the metadata information to determine the correct advice precedence according to our dynamic dispatch rule (see what follows), match again all JPIs against the code using the associated pointcuts, and insert appropriate advice-dispatch code at every join point shadow.

5.2. Implementation Based on abc

The current implementation of JPIs is restricted to a much simpler compilation scheme, imposed by the fact that the abc compiler does not support separate compilation. Following this restriction, we assume that all aspects, base code, and JPIs are present at the same time. Note that this has no impact on the modularity of type checking, but it does free us from dealing with metadata information in class files.

One detail that nevertheless requires further discussion is how we ensure the correct dispatch semantics for advices referring to JPIs. Remember that syntactically advice declarations do not at all refer to any pointcut. Instead they refer to a JPI declaration, which in turn may be bound to pointcuts by one or more **`exhibits`** clauses. To allow for maximal reuse of existing functionality in the abc compiler, we decided to implement the dispatch semantics through a transformation that computes for each such advice a single pointcut, based on the referenced JPI, its type hierarchy, and the **`exhibits`** clauses of those types. Let a be the advice to compute the pointcut for, as the set of other advices in the same aspect and es the set of all **`exhibits`** clauses in the program.

Then we compute the pointcut for a as follows.

$$\begin{aligned} pc(a, as, es) &= pc^{+}(a.jpi, es) \wedge \neg pc^{-}(a, as, es) \\ pc^{+}(jpi, es) &= \bigvee_{e \in es,\ e.jpi\ <:\ jpi} e.pc \\ pc^{-}(a, as, es) &= \bigvee_{a' \in as,\ a' \sqsubset a} pc^{+}(a'.jpi, es) \end{aligned}$$

The equation[10] for pc^{+} implements polymorphism: if a refers to $a.jpi$ then a will match not only on join points for $a.jpi$ itself but also for all subtypes. The equation for pc^{-} implements advice overriding within the same aspect: if an advice a' has the same kind as a but refers to a more specific JPI type, then a' overrides a, which means that a will not execute for the join points of this JPI. For advice that has been declared **final**, pc^{-} is simply skipped, so as to avoid overriding. The way in which we compute the pointcuts is quite similar to the way in which pointcuts are computed for advices associated with a JPT [Steimann et al. 2010, Section 4.3]; the main difference is the way in which we allow advice overriding.

To implement the previous equation, the implementation has to overcome a few technical obstacles. JPI declarations can rename formal arguments of their supertypes in their **extends** clauses. The implementation undoes this renaming in the back-end by inlining pointcuts. Further, the pointcut $pc^{-}(a, as, es)$ is used under negation. This raises an issue with argument-binding pointcuts, like `this(a)`, because they cannot be negated: if a pointcut does not match, there is no value that `a` could be bound to. Fortunately, abc supports a way to close such pointcuts so that the variables do not appear free any longer. This is done by rewriting a pointcut such as `this(a)` to `(λa.this(a))`; such a pointcut can be negated, and if `a` is of type `A`, the negation is equivalent to `!this(A)`, which yields the semantics we need.

Note that the separate compilation scheme from Section 5.1 would not restrict us from computing a single pointcut per advice, since at integration time the entire program with all JPIs and associated pointcuts is known. Again, the crucial point is that no error can happen at integration time.

5.3. Global Pointcuts

We implement the desired runtime semantics for global pointcut by inlining global pointcuts into every aspect and class, replacing occurrences of the `global(..)` pointcut by the corresponding global pointcut definition. In case a class contains no refinement for a global JPI (e.g., `void JP()`), and the class is not **sealed**, then we treat it as if the class (or aspect) actually defined an **exhibits** clause for the global JPI whose definition is simply `global()`, for instance, as follows.

```
exhibit void JP(): global();
```

5.4. Invariant Pointcut Designators

As noted in Section 4, we must ensure that join point interfaces are invariantly typed. As it turns out, in AspectJ it is not straightforward to ensure invariant typing for arguments. The problem is that, as we already explained in Section 2.1.3, the standard

[10] $\sqsubset$ denotes kind-specific subtyping for advices: $a' \sqsubset a$ means that a' and a are of the same kind and $a'.jpi < a.jpi$.

pointcuts **this**, **target**, and **args** come with a variant semantics. The fact that a JPI's arguments are typed invariantly against their associated advices does not change this fact; to enforce invariance, we also need to restrict the join point matching accordingly. One way to address this problem is to redefine the matching semantics of the **this**, **target**, and **args** pointcuts such that they match a join point only if the declared type at the join point is exactly the same as the declared type used within the pointcut. This design, however, would give up backward compatibility with AspectJ. Since our overall language design can be integrated as a fully backward-compatible extension, we opted for another design, such that existing AspectJ applications can be easily migrated.

The current implementation supports three additional pointcuts—namely **This**, **Target**, and **Args**, which have an invariant matching semantics. In the example, the pointcut.

```
pointcut checkingOut(double price, User usr):
  execution(∗ ShoppingSession.checkOut(..)) && Args(∗, price, ∗, usr);
```

would simply not match the `checkOut` method from Listing 1 because `checkOut` would assign `usr` a value of type `Customer`, not `User`. The pointcuts only match if the declared type of the respective argument position of the JPI is the same as the respective declared type at the shadow. (Note that we still match variantly on the actual runtime type, i.e., the pointcuts are allowed to match runtime values of subtypes of the shadow's respective declared type. This is safe because due to the invariance on declared types we know that the shadow's context can handle those subtypes.) Due to the introduction of these novel pointcuts, the existing semantics of **this**, **target**, and **args** remains unchanged. When programmers use one of those pointcuts within an **exhibits** clause, the compiler issues a warning, notifying the programmer that **This**, **Target**, or **Args** should be used instead, as otherwise type soundness cannot not be guaranteed. When using an invariant pointcut with a type variable, the implementation falls back to the default semantics of AspectJ for the respective pointcut, matching variantly again (up to the upper bound of the type variable).

5.5. Static Overloading

In addition to overriding, we also support static overloading of JPIs. For instance, one can write the following definition.

```
jpi void CheckingOut(double price, Customer cus);
jpi void CheckingOut(double price, int amount, Customer cus);
```

Similar to overloaded methods in Java, overloading of JPIs is resolved completely at compile time. Overloaded JPI definitions are thus treated exactly as if they had different names. Therefore, overloading is just a means to allow the programmer to document that two JPIs are inherently related; it has no semantic implications.

5.6. Reuse of Implementation for Closure Join Points

Interestingly, our abc extension for join point interfaces extends and completely reuses the original implementation for closure join points [Bodden 2011]. abc uses the JastAdd [Ekman and Hedin 2007] compiler front-end, which allows for a truly modular language definition. This allows the JPI extension to modularly define how the closure join points implementation needs to be refined to match the correct syntax and semantics that they require when used in combination with join point interfaces.

6. EMPIRICAL EVALUATION

We first discuss the benefits of join point interfaces based on previous studies. Then we report on a case study where we inspected several AspectJ applications for their usage of pointcuts, and converted two of them to use JPIs. To measure the impact of generic JPIs and controlled global quantification, we converted each project three times. The first version uses neither generics nor global JPIs, the second includes generics, and the third uses the full set of language mechanisms.

6.1. Benefits of Join Point Interfaces in Light of Previous Studies

Join point interfaces establish a clear contract between base code and aspects, aiding separate development, sharing ideas with XPIs and JPTs. The benefits of XPIs and JPTs on modularity have been empirically demonstrated in previous editions of TOSEM [Sullivan et al. 2010; Steimann et al. 2010].

Recently, Dyer et al. [2012] report on an exploratory study of the design impact of different approaches to aspect interfaces. They consider the evolution of aspect-oriented software using different approaches, namely plain AspectJ, annotation style, and quantified, typed events [Rajan and Leavens 2008]. While they do not consider JPIs in their study, their key results support our design of join point interfaces.

—The use of inter-type declarations is prevalent. By integrating JPIs in AspectJ, we inherit this mechanism for free.
—Quantification failure due to the need to advise join points that cannot be denoted using the pointcut language occurred on 5% of advised join point shadows. Explicit announcement addresses these cases nicely.
—The lack of quantification support with quantified typed events was problematic because it required to keep track of design rules that affect all members of a given class (e.g., make all methods **`synchronized`**). Implicit announcement with **`exhibits`** clauses supports these cases.
—Almost a fifth of changes to pointcuts was due to the fragile pointcut problem. In our setting, pointcuts would likely be per class, making those changes more localized.
—There were several instances where the fact that context information is restricted to join-point-specific attributes (**`this`**, **`target`**, **`args`**) was problematic, yielding additional complexity. Explicit announcement approaches make it easy to expose arbitrary context information.[11]

To evaluate the importance of a sound treatment of checked exceptions in aspect interfaces, it is instructive to look at the lessons learned by Robillard and Murphy in an effort to design robust programs with exceptions [Robillard and Murphy 2000]. They report that focusing on specifying and designing the exceptions from the very early stages of development of a system is not enough; exception handling is a global phenomenon that is difficult and costly to fully anticipate in the design phase. Thus, inevitably, the exceptions that can be thrown from modules are bound to evolve over time, as development progresses and this global phenomenon is better understood. The support that JPIs provide to report exception conformance mismatch between aspects and advised code in a modular fashion is therefore particularly necessary: as modules change their exception interface, immediate and local feedback is crucial to decide if these changes must be promoted to the actual contract between aspects and advised code. This avoids errors before system integration time.

[11]Context-aware aspects [Tanter et al. 2006] are a general mechanism that supports arbitrary context information with implicit announcement; a similar flexibility is found in AspectScript [Toledo et al. 2010].

6.2. Case Study Subjects

To find a practical semantics for JPIs, and in particular join point polymorphism, we have first inspected a set of existing AspectJ applications from the corpus of Khatchadourian et al. [2009]: AJHotDraw, an aspect-oriented version of JHotDraw, a drawing application; Glassbox, a diagnosis tool for Java applications; SpaceWar, a space war game that uses aspects to extend the game in various respects; and LawOfDemeter, a small set of aspects checking the compliance to the Law of Demeter programming rules [Lieberherr et al. 1988]. The code for all subjects is available from the JPI project Web site.

The first three projects were selected because of their comparatively large size and number of aspects. LawOfDemeter is a rather small project that showcases an interesting use of pointcuts, as discussed further shortly.

6.3. Join Point Polymorphism

We inspected the programs using both the AspectJ Development Tools [Clement et al. 2003] and AspectMaps [Fabry et al. 2011]. These tools allowed us to easily identify which advices advise which join point shadows. In particular, we focused on the shadows that are advised by more than one advice, as this hints at potential for subtyping. We also systematically investigated all pointcut expressions used in these projects and looked for potential type hierarchies. Our investigation revealed several interesting example hierarchies and clearly supports the usefulness of our semantics of join point subtyping. We now discuss a few representative examples.

6.3.1. Subtyping Patterns. We identify two patterns that programmers use to "emulate" subtyping with pointcuts. LawOfDemeter contains the following pointcuts.

```
pointcut MethodCallSite(): scope() && call(* *(..));
pointcut MethodCall(Object thiz, Object targt):
   MethodCallSite() && this(thiz) && target(targt);
pointcut SelfCall(Object thiz, Object targt):
   MethodCall(thiz,targt) && if(thiz == targt);
```

These pointcuts form an instance of a pattern that we call *subtype by restriction*. `MethodCall` restricts the join points exposed by `MethodCallSite` to instance methods, through additional **`this`** and **`target`** pointcuts. `SelfCall` restricts this set further by identifying self-calls using an additional **`if`** pointcut. A programmer could model this join point type hierarchy with JPIs as follows.

```
jpi Object MethodCallSite();
jpi Object MethodCall(Object thiz, Object targt) extends MethodCallSite();
jpi Object SelfCall  (Object thiz, Object targt) extends MethodCall(thiz,
    targt);
```

The example shows that it is useful to allow subtypes to expose more arguments than their supertypes (a.k.a. breadth subtyping): `MethodCall` exposes `thiz` and `targt`, while `MethodCallSite` exposes nothing at all.

SpaceWars includes various instances of the *subtype by restriction* pattern, but also features instances of the dual pattern, *supertype by union*. Consider the following.

```
pointcut syncPoint():
        call(void Registry.register(..))   ||
        call(void Registry.unregister(..)) ||
```

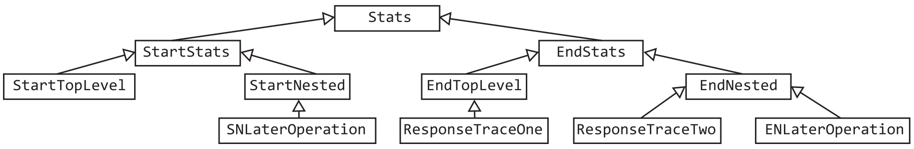


Fig. 4. A potential hierarchy of join point interfaces in Glassbox.

```
        call(SpaceObject[] Registry.getObjects(..)) ||
        call(Ship[] Registry.getShips(..));
pointcut unRegister(Registry registry):
        target(registry) &&
        (call(void register(..)) || call(void unregister(..)));
```

Here the pointcut `unRegister` matches a subset of the join points matched by `syncPoint` because `syncPoints` includes additional join points by disjunction (set union). Here also, the subtype induced by `unRegister` exposes an additional argument.

6.3.2. Depth of Subtyping Hierarchies. Glassbox proved to be a very interesting case study in that it provides over 80 aspects, and more than 200 pointcut definitions, with potential for nontrivial join point type hierarchies. Here, we only show one of the most interesting examples in Figure 4. The figure shows a hierarchy formed by 11 pointcuts within the aspect `ResponseTracer`. We have added `Stats` as a root type that the aspect does not contain explicitly, but which could be introduced to abstract the common parts of the `StartStats` and `EndStats` pointcuts.

6.3.3. Advice Overriding. Glassbox showcases the interest of being able to declare some advice as **`final`** to avoid overriding. An aspect in charge of system initialization advises the execution of `TestCase` object constructors. Five test classes implement the `InitializedTestCase` interface (`ITC` for short), and five implement the `ExplicitlyInitializedTestCase` interface (`EITC` for short); one class implements both (these interfaces are added via inter-type declarations). The aspect defines four before advices on these constructors, discriminating between different categories using pointcuts. These pointcuts correspond to four join point types $T_1 \ldots T_4$, where T_1 is a supertype of the three others. Without declaring the advice for T_1 final, the advice for T_1 never executes since it is always overridden. The only solution would then be to refactor the program to move the advice for T_1 in a separate aspect. Declare the advice for T_1 as **`final`** solves the problem in a simpler manner, as it guarantees that the advice will not be overridden and therefore always execute.

6.4. Join Point Polymorphism vs. Join Point Duplication

As discussed earlier, JPIs differ from JPTs in the way in which they resolve advice dispatch. Our design decision to implement join point polymorphism rather then duplication is reinforced by the case study. With JPTs, in Glassbox, for all classes that implement only one of the two interfaces `ITC` and `EITC`—10 out of 11—the advice associated with T_1, which initializes a factory object, is executed twice. Our dispatch semantics of JPIs avoids this problem: a given advice is guaranteed to execute *at most once* for any given join point (Section 4.3).

6.5. Per-Kind Advice Overriding

AJHotDraw contains the following definitions.

```
pointcut commandExecuteCheck(AbstractCommand acommand) :
        this(acommand)
        && execution(void AbstractCommand+.execute()) ..
        && !within(∗..JavaDrawApp.∗);
before(AbstractCommand acommand):
        commandExecuteCheck(acommand) {..}
pointcut commandExecuteNotify(AbstractCommand acommand) :
        commandExecuteCheck(acommand)
        && !within(org.jhotdraw.util.UndoCommand) ..
        && !within(org.jhotdraw.contrib.zoom.ZoomCommand);
after(AbstractCommand acommand):
        commandExecuteNotify(acommand) {..}
```

This is another instance of the *subtype by restriction* pattern, with `commandExecuteNotify` refining the pointcut `commandExecuteCheck`. This example validates our semantics to consider advice kinds separately when resolving advice overriding (Section 4.3). In the example, the first pointcut is advised with a before advice, while the second is advised by an after advice. Assume now that we had abstracted from those pointcuts using JPIs as in the following code.

```
before CheckingView(AbstractCommand acommand){..}
after NotifyingView(AbstractCommand acommand){..}
```

In this example, `NotifyingView` is a subtype of `CheckingView`. If we did *not* separate advices by advice kind when determining advice overriding then only the `NotifyingView` would execute at a `NotifyingView` join point, leading to an altered semantics compared to the original AspectJ program. Conversely, because we *do* separate advices by kind, when encountering a `NotifyingView` join point, the `CheckingView` advice is executed before the join point and the `NotifyingView` afterward.

6.6. Duplication of Advice due to the Lack of Generic Types

The typing rules for JPIs enforce invariance on both return and exception types of join point interfaces (Section 4.2). This is the simplest way to ensure soundness, but can be too rigid unless generic types are used. In an application like AJHotDraw, in which most pointcuts are very specific anyway, this rigidity is less problematic than in applications that rely on wide-matching pointcuts, such as LawOfDemeter. To assess the usefulness of generic JPIs, we first converted AJHotDraw and LawOfDemeter to use JPIs but not generics (and no global quantification either). All converted projects are available online on the JPI project Web page. In AJHotDraw, we found that only three advices required multiplication, but many copies had to be created, increasing the total number of advices in this project from 49 to 77. In LawOfDemeter the number of advices was increased more than 10 times, moving from 6 to 68!

To see why, let us consider the `Check` aspect defined in the LawOfDemeter (LoD) project, which checks that an object only sends messages to a certain set of closely related objects according to the Law of Demeter [Lieberherr et al. 1988]. Figure 5 shows an advice that registers the LoD violations within an aspect called `Check`. The important thing to note in this aspect definition is the extreme quantification used (`call(∗ ∗(..))`), which is typical of dynamic analysis aspects.

To migrate the `Check` aspect to use JPIs, conversely to what we suggested in Section 6.3.1, we *cannot* just use a single JPI like the following.

```
jpi Object MethodCall(Object thiz, Object targt);
```

The reason is that, in order to ensure type soundness, such a JPI could only match join points where the argument and return type at the shadow is `Object`. This means that

```
aspect Check {
        private IdentityHashMap objectViolations = new IdentityHashMap();

        public pointcut scope():
          !within(lawOfDemeter..*) && !cflow(withincode(* lawOfDemeter..*(..))) ;

        pointcut methodCalls(Object thiz, Object targt) :
           scope() && call(* *(..)) && this(thiz) && target(targt);

        after(Object thiz,Object targt): methodCalls(thiz,targt) {
                if (!ignoredTargets.containsKey(targt) &&
                    !Pertarget.aspectOf(thiz).contains(targt)) {
                        objectViolations.put(thisJoinPointStaticPart,
                                             thisJoinPointStaticPart);
                }
        }
}
```

Fig. 5. LoD check aspect (excerpt).

```
aspect Check {
        void registerViolation(Object thiz, Object targt, JoinPoint.StaticPart jp) {
                if (!ignoredTargets.containsKey(targt) &&
                     !Pertarget.aspectOf(thiz).contains(targt)) {
                        objectViolations.put(jp, jp);
                }
        }
        // one advice per type combination...
        after MethodCall(TemporalQueue thiz, TemporalQueue targt) {
                this.registerViolation(thiz, targt, thisJoinPointStaticPart);
        }
        after MethodCall(TemporalQueue thiz, TQ_N targt) {
                this.registerViolation(thiz, targt, thisJoinPointStaticPart);
        }
        after MethodCall(TemporalQueue thiz, TemporalQueue targt) {
                this.registerViolation(thiz, targt, thisJoinPointStaticPart);
        }
        // etc.
}
```

Fig. 6. LoD check aspect with (nongeneric) JPIs.

we would need to define one JPI per possible combination of types for **this**, **target**, and the return type! (Normall, even all possible types of checked exceptions would need to be considered, but the LoD project uses no checked exceptions.) In the concrete LoD project, just handling the `Check` aspect for the examples included in the project required us to write 21 JPIs.

```
jpi void    MethodCall(TemporalQueue thiz, TemporalQueue targt);
jpi int     MethodCall(TemporalQueue thiz, TQ_N targt);
jpi PQ_Node MethodCall(TemporalQueue thiz, TemporalQueue targt);
//etc.
```

In addition to this tedious and fragile list of JPI definitions, again due to invariance, the `Check` aspect itself has to contain one advice for each such JPI. As Figure 6 shows, all advice bodies are exact copies (note that for **after** advice, the return type of the JPI

```
aspect Check {
      // ... registerViolation ...
       <T, U> after MethodCall(T thiz, U targt){
           this.registerViolation(thiz, targt, thisJoinPointStaticPart);
      }
}
```

Fig. 7. LoD check aspect with a generic JPI.

Table II. Number of Advices Defined in Each Version

	AspectJ	Non-generic JPI	Generic JPI
LoD	6	68	6
AJHotDraw	49	77	49

is simply ignored). While this approach is “correct” in that all expected join points are matched and type soundness is preserved, it is highly cumbersome for programmers and, most importantly, it does not scale. As soon as one wants to apply the LoD aspects to other projects, more JPIs and advices have to be defined; the crosscutting concern is not modularized any longer.

It is worthwhile noting that JPTs would not suffer from this kind of duplication, due to their more flexible but unsound type system.

6.7. Generic Types to the Rescue

As explained earlier, our answer to this problem is the support for generic JPIs. Using generics, the preceding definitions can all be collapsed into one.

```
<R, A, B> jpi R MethodCall(A thiz, B targt);
```

This generic interface abstracts away the specific types involved. Now the programmer can define a generic version of the `Check` aspect using a polymorphic advice (Figure 7).

To assess the impact of generic JPIs, we reimplemented the AJHotDraw-JPI and LoD-JPI projects. As Table II shows, generic JPIs completely eliminate the problem of repeated advice declarations in both projects. While the number of advice declarations in the (nongeneric) JPI version was considerably increased (more than 10x for LoD), the version using generic JPIs presents the exact same number of advices. This clearly validates the practical positive impact of extending JPIs to support parametric polymorphism.

6.8. Duplication of Exhibits Clauses due to Absence of Global Quantification

As our study also revealed, if JPIs are used in combination with highly crosscutting aspects, developers may be forced to modify several (if not all) existing classes to introduce **`exhibits`** clauses. To illustrate, let us consider the situation that a programmer wishes to apply the generic JPI version of the `Check` aspect (Figure 7) to two classes `TemporalQueue` and `Repository`. The programmer would introduce the **`exhibits`** clauses, as shown in Figure 8. We note that this approach is fragile and does not scale: programmers must modify every single class to introduce the **`exhibits`** clauses. Moreover, this redundancy is unnecessary, as in most cases all classes will bind the same JPI to the same pointcut expression. In the migration of LoD to its generic JPI version, this problem forced us to modify 21 of 23 classes to include the corresponding **`exhibits`** clauses. JPTs, in this case, would suffer from the same duplication.

```
import jpis.*;

public class TemporalQueue extends PriorityQueue
{
  <R, T, I> exhibits R MethodCall(T thiz, I targt) :
    call(R *(..))
    && This(thiz)
    && Target(targt)
    && !within(lawOfDemeter..*)
    && !cflow(withincode(* lawOfDemeter..*(..)));
  ...
}

public class Repository extends Entity
{
  <R, T, I> exhibits R MethodCall(T thiz, I targt) :
    call(R *(..))
    && This(thiz)
    && Target(targt)
    && !within(lawOfDemeter..*)
    && !cflow(withincode(* lawOfDemeter..*(..)));
  ...
}
```

Fig. 8. Base code advised by the generic JPI version of check aspect.

Table III. Number of Exhibits Clauses Defined in Each Version

	Generic JPI version	Global JPI version
LoD	130	0
AJHotDraw	46	46

6.9. Controlled Global Quantification to the Rescue

Using a global JPI allows programmers to easily replace all **exhibits** clauses by the following declaration.

```
  <T, U, V> global jpi T MethodCall(U thiz, V targt) :
      call(T *(..))
      && This(thiz)
      && Target(targt);
      //&& !within(lawOfDemeter..*)
      //&& !cflow(withincode(* lawOfDemeter..*(..)));
```

Note that the commented-out clauses in the previous pointcut become unnecessary when we seal all classes/aspects in the `lawofdemeter` package. Although sealing is not generally enough to avoid infinite recursion in the execution of aspects (see Section 7 for details), it is in the case of LoD, as the LoD aspects call only code from the `lawofdemeter` package and from the JDK, which in our compiler is sealed by default. AJHotDraw uses no global pointcuts, and therefore also cannot benefit from sealing. For both applications it would not have made sense to seal all types by default.

To assess the impact of global pointcuts, we reimplemented the generic JPI versions of both AJHotDraw and LoD with global pointcuts. As Table III shows, global pointcuts significantly decrease the amount of scattered **exhibits** clauses in projects such as LoD, which implement highly crosscutting aspects. While the number of **exhibits** clauses in the (generic) JPI version of LoD was 130, the version using global JPIs in

```
joinpointtype Printing allows after {}

class Util exhibits Printing {
  pointcut Printing: call(* print*(..));

  public static void main(String args[]) { Util.print("Hello"); }

  public static void print(String s) { System.out.println(s); }
}

aspect Asp advises Printing {
  after(Printing Printing) { Util.print(" World"); }
}
```

Listing 6. JPT example causing infinite loop.

fact goes without any **exhibits** clauses. As expected, however, for applications such as AJHotDraw, which comprise highly specific pointcuts, global quantification cannot help, as different classes must expose different join point shadows for matching to the respective aspects.

There is a clear tension between global quantification and class-level **exhibits** clauses. On the one hand, **exhibits** declarations at the class level demand more code annotations but have the positive effect of allowing for truly local reasoning at the class level. On the other hand, global pointcuts allow programmers to eliminate many code annotations at the cost of losing the ability to reason about advising locally.

Our case study shows that none of the two mechanisms is ideal for every aspect; highly crosscutting aspects do benefit from global quantification. Many of them, like LoD, or more generally dynamic analyses, cause no harm, because they are only observing the base-code execution. For other aspects that are meant to directly affect the base execution, global quantification may be the wrong choice. The design we propose reflects the intent to be pragmatic and support both families of aspects: programmers can make the choice of which mechanism to use, in well-typed setting.

7. RELATED WORK

In Section 2, we discussed previous approaches to aspect decoupling. We now give additional details on join point types, and describe how our work relates to event-based publish/subscribe systems. Finally, we discuss other approaches to type-safe aspects.

7.1. Further Information on Join Point Types

In Section 2.3.1 we have elaborated on the facets of JPTs that are most relevant to this work. We now discuss additional information about other minor differences with respect to join point interfaces.

Symmetry. In our proposal, both classes and aspects are allowed to exhibit join points. This is different from JPTs, where aspects cannot do so. In other words, in the semantics of JPTs, aspects are always sealed. According to Steimann et al. [2010, page 10], the motivation for this asymmetric approach is to avoid the possibility of infinite recursion (caused by an advice advising itself). Unfortunately, only ensuring that aspects do not *lexically* emit join points is not enough. Consider the JPT code in Listing 6. In this example, the aspect itself indeed does not expose any join points. However, it invokes `Util.print`, which does. This triggers an infinite recursion.

Tanter showed earlier that infinite recursion can be avoided by introducing execution levels [Tanter 2010]. Execution levels are a *dynamic* property of control flow: the entire control flow of an aspect is prevented from being advised by itself, not just the join points that are lexically raised within the aspect. An efficient AspectJ implementation of execution levels has already been developed [Tanter et al. 2010], showing how multiple dynamic analyses aspects can be reused off-the-shelf, even when the whole Java standard libraries are open to advising. We plan to study the integration of execution levels and JPIs in the future.

First-Class Join Points and Representation. Join points are first-class objects in JPTs but not in JPIs. In that respect, we simply decided to follow the design of AspectJ, in which join points are not first class either. In any case, just like in AspectJ, an advice can obtain a reified join point mirror through reflection using **`thisJoinPoint`**. Other aspect languages, like AspectScheme [Dutchyn et al. 2006], also support first-class join points, although this is arguably more natural in these languages where functions are first class. First-class join points are more powerful because they can outlive their original execution context.

Whether first class or not, the important point for the purpose of sound decoupling is that interfaces must carry the appropriate type information to allow modular type checking. Just like first-class functions can be statically typed, first-class join points are not problematic per se. In the design of JPTs, however, join point types miss crucial typing information about return types and checked exceptions. While one could add this information to the record-like representation of JPTs, one would in essence obtain the description of a typed signature of a first-class function, but with a different syntax. To us, the fact that type information is needed about the return type and exception types in order to soundly specify the interface at a join point strongly suggests that a method signature represention is better suited than a record definition. Interestingly, other approaches to type-safe aspects (discussed later in Section 7.3) also adopt a method/function type to describe join points.

Semantics of Explicit Announcement. Finally, the explicit join points used in JPTs can have a somewhat surprising semantics to Java programmers, as they introduce a form of dynamic scoping. This issue also leads to the potential to have data races on local variables if aspects decide to execute the original join point asynchronously. Earlier work by Bodden discusses these problems in detail; closure join points were designed to address these issues [Bodden 2011].

7.2. Event-Based Languages

There is a large body of work on expressive language mechanisms for event announcement, composition, and processing, usually called event-based languages, or also summarized under the umbrella term publish/subscribe. In general, such systems differ from join point interfaces in that they typically focus more on the event handling side than on the event announcement side. With join point interfaces, the emphasis is on supporting both implicit and explicit announcement in a uniform, flexible, and yet type-safe manner. Event-based languages typically allow for explicit announcement only, but often provide for more declarative event-processing facilities. In contrast, advices in AspectJ contain plain Java code, which makes them anything but declarative. But there is also a conceptual difference. Pure event-based systems typically have a notion of *atomic* events. This is very different from AspectJ, in which join points embed *computation*, and thus have a duration, can be skipped, repeated, or replaced by an alternative computation. This embedding of computation inside join points largely complicates the control flows and data flows involved, and makes static typing more challenging. To see how, we next discuss how JPIs relate to five existing event-based approaches in this area, namely EJP, Ptolemy, EventJava, EScala, and Fabric.

Explicit Join Points (EJP). Hoffman and Eugster [2007] first introduced the idea of trading obliviousness such that advised code raises events explicitly. Explicit Join Points, or EJPs, support a novel paradigm called cooperative aspect-oriented programming [Hoffman and Eugster 2012]. EJPs only support explicit announcement and are close to AspectJ in the sense that they rely on the original flexibility and unsound typing rules for around advice. Through the language design of EJPs, the interaction at those program points can be very rich. A particularly interesting feature is that the EJP implementation allows an advice to consume checked exceptions (by handling them with a `try/catch` block). As the authors show, this can be a very useful feature which can often simplify code readability and maintenance. Unfortunately, though, this semantics is only correct under the assumption that some aspect actually will be connected to each such EJP at any time. The EJP language addresses this issue by allowing EJP declarations to define a `handles` clause, which indicates that the EJP is guaranteed to handle certain checked exception types. The compiler assumes a closed program, checking that indeed some advice is bound to such an EJP. It also checks that the execution of the advice does not depend on dynamic checks, because if it did there may be an execution path on which a checked exception evades the EJP without being handled. This semantics, while interesting and flexible, is therefore not a semantics that we can adopt for JPIs, because we want to free base-code programmers from any dependency about specific aspects, and want to ensure that type checking can be carried out modularly. On a different topic, quite interestingly, the case study of Hoffman and Eugster shows good use cases for generic types even in the absence of global quantification. In our study we only found two nonglobal JPIs for which generic types were helpful.

Ptolemy. Like EJPs, Ptolemy [Rajan and Leavens 2008] also only supports explicit announcement of events. Events are defined with event types, which syntactically follow JPTs in that they resemble records. Like JPTs, they miss information about checked exceptions, but unlike JPTs they do include a return type. It is quite interesting to compare Ptolemy to JPIs because it shows what typing rules can be relaxed if one restricts the expressive power of advices. Ptolemy does support around advice, but forces around advice to always proceed with the original context values (arguments, receiver). A recent publication on Ptolemy explains the support for depth subtyping that is possible thanks to this restriction [Fernando et al. 2012]: for instance, if one event type binds a value of type `List`, then an event subtype can safely specialize this type to `LinkedList`. This is safe because the type is only used to pass values into the advice but not from the advice back into the advised context. Further, because Ptolemy only supports explicit announcement, emitted join points have a single most specific type, simplifying advice dispatch. Ptolemy also supports behavioral contracts, called translucid contracts [Bagherzadeh et al. 2011], to specify and verify control effects induced by event handlers. These verification techniques go beyond usual type checks, but would be interesting to see in combination with JPIs. Ptolemy is clearly lacking some better handling of exceptions. Code drawn from a recent case study by the authors [Dyer et al. 2012] shows that event types do not declare exceptions, but that all advices declare to throw `Throwable`. Nevertheless, none of the contexts that raise the respective explicit join points is actually prepared to catch such exceptions. This approach is therefore unsound with respect to the Java semantics of checked exceptions.

EventJava. The language EventJava by Eugster and Jayaram [2009] focuses even more on event processing than announcement. The language allows for powerful event combinators and filters, and a notion of “streams”, which give programmers a view on a sliding window over a sequence of equally typed events. EventJava is quite different from JPIs, though, in that all its event handling occurs asynchronous to the base program. Event handlers therefore cannot return values. There is also no facility for implicit announcement.

Distributed Programing with Typed Events. In a recent piece of work on distributed programing with typed events, Eugster and Guerraoui [2004] show how typed events can successfully establish high-level abstractions in a distributed setting. Interestingly, the authors discuss not only a language-based but also a library-based approach. From a type-checking perspective, the scenario appears similar to EventJava, as events are explicitly announced and always asynchronous.

Fabric. Fabric [Liu et al. 2009] is a Java language extension that supports distributed programming in a largely transparent fashion. Programmers can access worker nodes, dissemination nodes, and storage nodes through special object handles. Invocations on these nodes are automatically processed through the language runtime without requiring programmers to write boilerplate code. Event announcement happens through standard method calls and is thus always explicit. One feature that sets Fabric apart from most other languages is a type system that allows programmers to track information flow throughout the distributed system. It would be interesting to think about the impact of adding such a feature to the JPI language.

EScala. EScala [Gasiunas et al. 2011] is an approach to modular event-driven programming in Scala, which, like JPTs and JPIs, also combines implicit and explicit events. EScala does not support **around** advice, so event definitions need not declare return types; exception types are missing, but this reflects the design philosophy of the Scala language. EScala treats both events and handlers as object members, subject to encapsulation and late binding. Aspects specify events of interest with respect to specific source objects. This contrasts with both JPIs and most other approaches, where advices are attached to event types, regardless of the sources that emit these events.

7.3. Type Soundness and Aspects

Soundness issues with the type system of AspectJ were first reported by Wand et al. [2004], although no solution was proposed. Jagadeesan et al. [2006] formulate a sound approach in which an advice type may depend on explicitly declared type variables. Like in our approach, they use the same signature for **proceed** and the corresponding advice, and therefore require invariance as well. Some flexibility is regained because the type variables from the signatures can be instantiated for each join point. Due to parametricity, it is difficult to express arbitrary replacement advice.

MiniMAO$_1$ [Clifton and Leavens 2006] and StrongAspectJ [De Fraine et al. 2008] both consider different signatures for advice and **proceed**, thereby allowing more liberal pointcut/advice bindings while maintaining soundness. StrongAspectJ is more flexible in that it supports signature ranges for pointcuts and type variables for generic advices. As recognized by the authors, however, the more expressive typing constructs of StrongAspectJ result in quite complicated syntax forms. The typig rules for JPIs are more restrictive (our type ranges are basically singletons) but this design gives us the advantage of being able to offer a simpler syntax. The A calculus [De Fraine et al. 2010] refines the ideas of StrongAspectJ and proposes a simpler syntax, and a more flexible typing approach in which both the advice body and **proceed** are given the same fresh type variable, which fits in the declared type range.

Aspectual Caml [Masuhara et al. 2005] is an aspect-oriented extension of OCaml. An interesting feature of Aspectual Caml is that it uses type information to influence matching, rather than for reporting type errors. More precisely, they infer the type of pointcuts from the associated advices, and only match join points that are valid according to these inferred types. It is unclear if the inference approach may be ported to an object-oriented language with subtype polymorphism.

All the aforesaid approaches are formulated in a traditional pointcut-advice setting, without relying on interfaces to uncouple advised and aspect code. Our design of generic JPIs basically follows the proposal of Jagadeesan et al. This choice is pragmatic: while we have clear evidence of the interest of this approach for JPIs in our case study, we have not faced a case where the extra flexibility of StrongAspectJ or the A calculus was required. This may change in the future as we experiment with more programs.

8. CONCLUSION

Join Point Interfaces (JPIs) support flexible and safe decoupling of aspect-oriented programs through modular type checking. Like interfaces in statically typed object-oriented languages, JPIs are a machine-checked type-based contract that supports separate development in a robust and safe manner. Key to this support is the specification of JPIs as method-like signatures with return types and checked exception types. JPIs can be organized in hierarchies to structure the space of join points in a flexible manner, enabling join point polymorphism and dynamic advice dispatch. The use of parametric polymorphism in JPI definitions, along with support for controlled global quantification, makes it possible for JPIs to preserve most of the flexibility that AspectJ offers, without sacrificing type safety.

We have implemented JPIs as a publicly available AspectJ extension in the abc compiler, and have rewritten a set of existing AspectJ programs to take advantage of JPIs. This study supports our major design choices. In particular, the study confirms that the design is both flexible and type safe. Further experiments will tell if more advanced typing mechanisms like type ranges [De Fraine et al. 2008] are needed in practice. We also plan to investigate to what extent join point interfaces can be enriched with richer behavioral specifications.

ACKNOWLEDGMENTS

We wish to thank Friedrich Steimann and the anonymous reviewers for many helpful comments on earlier versions of this article, which greatly helped us improve the quality of the presentation.

REFERENCES

J. Aldrich. 2005. Open modules: Modular reasoning about advice. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*. A. P. Black, Ed., Lecture Notes in Computer Science, vol. 3586, Springer, 144–168.

P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. De Moor, D. Sereni, G. Sittampalam, and J. Tibble. 2005. ABC: An extensible aspectj compiler. In *Proceedings of the 4th ACM International Conference on Aspect-Oriented Software Development (AOSD'05)*. ACM Press, New York, 87–98.

M. Bagherzadeh, H. Rajan, G. T. Leavens, and S. Mooney. 2011. Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD'11)*. ACM Press, New York, 141–152.

E. Bodden. 2011. Closure joinpoints: Block joinpoints without surprises. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD'11)*. ACM Press, New York, 117–128.

G. Bracha, P. Ahe, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. 2010. Modules as objects in newspeak. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10)*. Lecture Notes in Computer Science, vol. 6183, Springer, 405–428.

A. Clement, A. Colyer, and M. Kersten. 2003. Aspect-oriented programming with ajdt. In *Proceedings of the ECOOP Workshop on Analysis of Aspect-Oriented Software.*

C. Clifton and G. T. Leavens. 2006. MiniMAO1: An imperative core language for studying aspect oriented reasoning. *Sci. Comput. Program.* 63, 312–374.

C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. 2000. MultiJava: Modular open classes and symmetric multiple dispatch in java. In *Proceedings of the 15th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'00)*. ACM Press, New York, 130–145.

B. De Fraine, E. Ernst, and M. Südholt. 2010. Essential aop: The a calculus. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10)*. 101–125.

B. De Fraine, M. Südholt, and V. Jonckers. 2008. StrongAspectJ: Flexible and safe pointcut/advice bindings. In *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD'08)*. ACM Press, New York, 60–71.

C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. 2006. Semantics and scoping of aspects in higherorder languages. *Sci. Comput. Program.* 63, 3, 207–239.

R. Dyer, H. Rajan, and Y. Cai. 2012. An exploratory study of the design impact of language features for aspect-oriented interfaces. In *Proceedings of the 11th International Conference on Aspect-Oriented Software Development (AOSD'12)*. É. Tanter and K. J. Sullivan, Eds., ACM Press, New York, 143–154.

T. Ekman and G. Hedin. 2007. The jastadd extensible java compiler. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA'07)*. 1–18.

P. Eugster and K. Jayaram. 2009. Eventjava: An extension of java for event correlation. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP'09)*. Lecture Notes in Computer Science, vol. 5653, Springer, 570–594.

P. T. Eugster and R. Guerraoui. 2004. Distributed programming with typed events. *IEEE Softw.* 21, 2, 56–64.

J. Fabry, A. Kellens, and S. Ducasse. 2011. Aspectmaps: A scalable visualization of join point shadows. In *Proceedings of 19th IEEE International Conference on Program Comprehension (ICPC'11)*. IEEE Computer Society Press, 121–130.

R. D. Fernando, R. Dyer, and H. Rajan. 2012. Event type polymorphism. In *Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages (FOAL'12)*. ACM Press, New York, 33–38.

V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé. 2011. EScala: Modular event-driven object interactions in scala. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD'11)*. ACM Press, New York, 227–240.

S. Gudmundson and G. Kiczales. 2001. Addressing practical software development issues in aspectj with a pointcut interface. In *Proceedings of the Workshop on Advanced Separation of Concerns.*

K. Gybels and J. Brichau. 2003. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd ACM International Conference on Aspect-Oriented Software Development (AOSD'03)*. M. Aksit, Ed., ACM Press, New York, 60–69.

E. Hilsdale and J. Hugunin. 2004. Advice weaving in aspectj. In *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD'04)*. K. Lieberherr, Ed., ACM Press, New York, 26–35.

K. Hoffman and P. Eugster. 2007. Bridging java and aspectj through explicit join points. In *Proceedings of the 9th International Symposium on Principles and Practice of Programming in Java (PPPJ'07)*. ACM Press, New York, 63–72.

K. Hoffman and P. Eugster. 2012. Trading obliviousness for modularity with cooperative aspectoriented programming. *ACM Trans. Softw. Engin. Methodol.* 22, 3.

A. Igarashi, B. C. Pierce, and P. Wadler. 2001. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.* 23, 3, 396–450.

M. Inostroza, É. Tanter, and E. Bodden. 2011. Join point interfaces for modular reasoning in aspect-oriented programs. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'11)*. New Ideas Track.

R. Jagadeesan, A. Jeffrey, and J. Riely. 2006. Typed parametric polymorphism for aspects. *Sci. Comput. Program.* 63, 267–296.

R. Khatchadourian, P. Greenwood, A. Rashid, and G. Xu. 2009. Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software. In *Proceedings of the International Conference on Automated Software Engineering (ASE'09)*. IEEE/ACM, 575–579.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. 2001. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*. J. L. Knudsen, Ed., Lecture Notes in Computer Science, vol. 2072, Springer, 327–353.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. 1997. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*. Lecture Notes in Computer Science, vol. 1241, Springer, 220–242.

G. Kiczales and M. Mezini. 2005. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*. ACM Press, New York, 49–58.

R. Laddad. 2003. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Press.

K. Lieberherr, I. Holland, and A. Riel. 1988. Object-oriented programming: An objective sense of style. In *Proceedings of the 3rd International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'88)*. N. Meyrowitz, Ed., ACM Press, New York, 323–334.

J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. 2009. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. ACM Press, New York, 321–334.

H. Masuhara, G. Kiczales, and C. Dutchyn. 2003. A compilation and optimization model for aspect-oriented programs. In *Proceedings of the 12th International Conference on Compiler Construction (CC'03)*. G. Hedin, Ed., Lecture Notes in Computer Science, vol. 2622, Springer, 46–60.

H. Masuhara, H. Tatsuzawa, and A. Yonezawa. 2005. Aspectual caml: An aspect-oriented functional language. In *Proceedings of the 10th ACM SIGPLAN Conference on Functional Programming (ICFP'05)*. ACM Press, New York, 320–330.

M. Odersky, L. Spoon, and B. Venners. 2008. *Programming in Scala: A Comprehensive Step-by-Step Guide*. Artima Inc.

N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. De Moor, and G. Sittampalam. 2006. Adding open modules to aspectj. In *Proceedings of the 5th ACM International Conference on Aspect-Oriented Software Development (AOSD'06)*. ACM Press, New York, 39–50.

A. Paepcke, Ed. 1993. *Object-Oriented Programming: The CLOS Perspective*. MIT Press.

D. Parnas. 1972. On the criteria for decomposing systems into modules. *Comm. ACM* 15, 12, 1053–1058.

H. Rajan and G. T. Leavens. 2008. Ptolemy: A language with quantified, typed events. In *Proceedings of the 22nd European Conference on Object-oriented Programming (ECOOP'08)*. J. Vitek, Ed., Lecture Notes in Computer Science, vol. 5142, Springer, 155–179.

J. C. Reynolds. 1983. Types, abstraction, and parametric polymorphism. *Inf. Process.* 83, 513–523.

M. Robillard and G. Murphy. 2000. Designing robust java programs with exceptions. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT/FSE'00)*. 2–10.

F. Steimann. 2006. The paradoxical success of aspect-oriented programming. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'06)*. ACM Press, New York, 481–497.

F. Steimann. 2013. Personal communication.

F. Steimann, T. Pawlitzki, S. Apel, and C. Kästner. 2010. Types and modularity for implicit invocation with implicit announcement. *ACM Trans. Softw. Engin. Methodol.* 20, 1, 1–43.

M. Stoerzer and J. Graf. 2005. Using pointcut delta analysis to support evolution of aspect-oriented software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*. 653–656.

K. Sullivan, W. G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari. 2010. Modular aspect-oriented design with xpis. *ACM Trans. Softw. Engin. Methodol.* 20, 2.

É. Tanter. 2010. Execution levels for aspect-oriented programming. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD'10)*. ACM Press, New York, 37–48.

É. Tanter, K. Gybels, M. Denker, and A. Bergel. 2006. Context-aware aspects. In *Proceedings of the 5th International Symposium on Software Composition (SC'06)*. W. Löwe and M. Südholt, Eds., Lecture Notes in Computer Science, vol. 4089, Springer, 227–242.

É. Tanter, P. Moret, W. Binder, and D. Ansaloni. 2010. Composition of dynamic analysis aspects. In *Proceedings of the 9th ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE'10)*. ACM Press, New York, 113–122.

R. Toledo, P. Leger, and É. Tanter. 2010. Aspectscript: Expressive aspects for the web. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD'10)*. ACM Press, New York, 13–24.

M. Wand, G. Kiczales, and C. Dutchyn. 2004. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.* 26, 5, 890–910.

Received May 2012; revised January 2013; accepted January 2013