A Typed Monadic Embedding of Aspects

Nicolas Tabareau, Ismael Figueroa, Éric Tanter

A Typed Monadic Embedding of Aspects

Nicolas Tabareau
ASCOLA Group
INRIA, France
nicolas.tabareau@inria.fr

Ismael Figueroa
PLEIAD Laboratory & ASCOLA Group
DCC University of Chile & INRIA
ifiguero@dcc.uchile.cl

Éric Tanter
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Chile
etanter@dcc.uchile.cl

Abstract

We describe a novel approach to embed pointcut/advice aspects in a typed functional programming language like Haskell. Aspects are first-class, can be deployed dynamically, and the pointcut language is extensible. Type soundness is guaranteed by exploiting the underlying type system, in particular phantom types and a new anti-unification type class. The use of monads brings type-based reasoning about effects for the first time in the pointcut/advice setting, thereby practically combining Open Modules and EffectiveAdvice, and enables modular extensions of the aspect language.

1. Introduction

Aspect-oriented programming languages support the modular definition of crosscutting concerns through a join point model [15]. In the pointcut/advice mechanism, crosscutting is supported by means of pointcuts, which quantify over join points, in order to implicitly trigger advice [35]. Such a mechanism is typically integrated in an existing programming language by modifying the language processor, may it be the compiler (either directly or through macros), or the virtual machine. In a typed language, introducing pointcuts and advices also means extending the type system, if type soundness is to be preserved. For instance, AspectML [6] is based on a specific type system in order to safely apply advice. AspectJ [14] does not substantially extend the type system of Java and suffers from soundness issues. StrongAspectJ [7] addresses these issues with an extended type system. In both cases, proving type soundness is rather involved because a whole new type system has to be dealt with.

In functional programming, the traditional way to tackle language extensions, mostly for embedded languages, is to use monads [22]. Early work on AOP suggests a strong connection to monads. De Meuter proposed to use them to lay down the foundations of AOP [21], and Wand *et al.* used monads in their denotational semantics of pointcuts and advice [35]. Recently, Tabareau proposed a weaving algorithm that supports monads in the pointcut and advice model, which yields benefits in terms of extensibility of the aspect weaver [30]. Nevertheles in this work, the weaver itself was not monadic but integrated internally in the system. This connection was exploited in recent preliminary work by the authors, to construct an extensible monadic aspect weaver, in the context of Typed Racket [11]. However, contrary to what the name suggests, the proposed monadic weaver was not fully typed because of limitations in the type system of Typed Racket.

This work proposes a lightweight, full-fledged embedding of aspects in Haskell, that is both typed and monadic. By *lightweight*, we mean that aspects are provided as a small standard Haskell library[1]. The embedding is *full-fledged* because it supports dynamic deployment of first-class aspects with an extensible pointcut language—as is usually found only in dynamically-typed aspect languages like AspectScheme [9] and AspectScript [33] (Section 2).

By *typed*, we mean that in the embedding, pointcuts, advices, and aspects are all statically typed (Section 3), and pointcut/advice bindings are proven to be safe (Section 4). Type soundness is directly derived by relying on the existing type system of Haskell (type classes [34], phantom types [16], and some recent extensions of the Glasgow Haskell Compiler). Specifically, we define a novel type class for anti-unification [25, 26], key to define safe aspects.

Finally, because the embedding is *monadic*, we derive two notable advantages over ad-hoc approaches to introducing aspects in an existing language. First, we can directly reason about aspects and effects using traditional monadic techniques. In short, we can generalize the interference combinators of EffectiveAdvice [23] in the context of pointcuts and advice (Section 5). Second, because we embed a monadic weaver, we can modularly extend the aspect language semantics. We illustrate this with several extensions and show how type-based reasoning can be applied to language extensions (Section 6). Section 7 discusses several issues related to our approach, Section 8 reviews related work, and Section 9 concludes.

2. Introducing Aspects

A premise for aspect-oriented programming in functional languages is that function applications are subject to aspect weaving. We introduce the term *open application* to refer to a function application that generates a join point, and consequently, can be woven.

Open function applications. Opening all function applications in a program or only a few selected ones is both a language design question and an implementation question. At the design level, this is the grand debate about *obliviousness* in aspect-oriented programming. Opening all applications is more flexible, but can lead to fragile aspects and unwanted encapsulation breaches. At the im-

É. Tanter is partially funded by FONDECYT project 1110051.
I. Figueroa is funded by a CONICYT-Chile Doctoral Scholarship.

[1] Available, with examples, at http://pleiad.cl/haskellaop

plementation level, opening all function applications requires either a preprocessor or runtime support.

For now, we focus on *quantification*—through pointcuts—and opt for a conservative design in which open applications are realized *explicitly* using the `#` operator: `f # 2` is the same as `f 2`, except that the application generates a join point that is subject to aspect weaving. We will come back to obliviousness in Section 7.3, showing how different answers can be provided within the context of our proposal.

Monadic setting. Our approach to introduce aspects in a pure functional programming language like Haskell can be realized without considering effects. Nevertheless, most interesting applications of aspects rely on computational effects (*e.g.* tracing, memoization, exception handling, etc.). We therefore adopt a monadic setting from the start. Also, as we show in Section 5, this allows us to exploit the approach of EffectiveAdvice [23] in order to do type-based reasoning about effects in presence of aspects.

Illustration. As a basic example, consider the following:

```
advice:
ensurePos proceed n = proceed (abs n)

monadic version of sqrt:
sqrtM n = return (sqrt n)

using an aspect:
program n = do deploy (aspect (pcCall sqrtM) ensurePos)
               sqrtM # n
```

The advice `ensurePos` enforces that the argument of a function application is a positive number, by replacing the original argument with its absolute value. We then deploy an aspect that reacts to applications of `sqrtM`, the monadic version of `sqrt`, by executing this advice. This is specified using the pointcut `(pcCall sqrtM)`. Evaluating `program -4` results in `sqrtM` to be eventually applied with argument `4`. As can be seen, aspects are created with `aspect` and deployed with `deploy`.

Our introduction of AOP therefore simply relies on defining aspects (pointcuts, advices), the underlying aspect environment together with the operations to deploy and undeploy aspects, and open function application.

The remainder of this section briefly presents these elements, and the following section concentrates on the main challenge: properly typing pointcuts and ensuring type soundness of pointcut/advice bindings.

2.1 Join Point Model

The support for crosscutting provided by a programming language lies in its *join point model* (JPM) [19]. A JPM is composed by three elements: *join points* that represents the points of a program that aspects can affect, a *means of identifying* join points—here, pointcuts—and a *means of effecting* at join points—here, advices.

Join points. Join points are function applications. A join point `JP` contains a function of type `a → m b`, and an argument of type `a`. `m` is a monad denoting the underlying computational effect stack. Note that this means that only functions that are properly lifted to a monadic context can be advised. In addition, in order for pointcuts to be able to reason about the type of advised functions, we require the functions to be `PolyTypeable`[2].

```
data JP m a b = (Monad m, PolyTypeable (a → m b)) ⇒
                JP (a → m b) a
```

From now on, we omit the type constraints related to `PolyTypeable` (the `PolyTypeable` constraint on a type is required each time the type has to be inspected dynamically; exact occurrences of this constraint can be found in the implementation).

Pointcuts. A pointcut is a predicate on the current join point. It is used to identify join points of interests. A pointcut simply returns a boolean to indicate whether it matches the given join point.

```
data PC m a b = Monad m ⇒ PC (∀ a' b'. m (JP m a' b' → m Bool))
```

A pointcut is represented as a value of type `PC m a b`. (`a` and `b` are used to ensure type safety, as discussed in Section 3.1.) The predicate itself is a function `∀ a' b'. m (JP m a' b' → m Bool)`, meaning it has access to the monadic stack. The `∀` declaration quantifies on type variables `a'` and `b'` (using rank-2 types) because a pointcut should be able to match against any join point, regardless of the specific types involved (we come back to this in Section 3.1).

We provide two basic pointcut designators, `pcCall` and `pcType`, as well as logical pointcut combinators, `pcOr`, `pcAnd`, and `pcNot`.

```
pcType f = let t = polyTypeOf f in PC (_type t)
           where _type t = return (\jp →
             return (compareType t jp))

pcCall f = let t = polyTypeOf f in PC (_call f t)
           where _call f t = return (\jp →
             return (compareFun f jp &&
                     compareType t jp))
```

`pcType f` matches all calls to functions that have a type compatible with `f` (see Section 3.1 for a detailed definition) while `pcCall f` matches all calls to `f`. In both cases, `f` is constrained to allow using the `PolyTypeable` introspection mechanism, which provides the `polyTypeOf` function to obtain the type representation of a value. This is used to compare types with `compareType`.

To implement `pcCall` we require a notion of function equality[3]. This is used in `compareFun` to compare the function in the join point to the given function. Note that we also need to perform a type comparison, using `compareType`. This is because a polymorphic function whose type variables are instantiated in one way is equal to the same function but with type variables instantiated in some other way (*e.g.* `id :: Int → Int` is equal to `id :: Float → Float`).

Users can define their own pointcut designators. For instance, we can define control-flow pointcuts like AspectJ's `cflow` (discussed briefly in Section 6), data flow pointcuts [18], pointcuts that rely on the trace of execution [8], etc.

Advice. An advice is a function that executes in place of a join point matched by a pointcut. This replacement is similar to open recursion in EffectiveAdvice [23]. An advice receives a function (known as the `proceed` function) and returns a new function of the same type (which may or may not apply the original `proceed` function internally). We introduce a type alias for advice:

```
type Advice m a b = (a → m b) → a → m b
```

For instance, the type `Monad m ⇒ Advice m Int Int` is a synonym for the type `Monad m ⇒ (Int → m Int) → Int → m Int`. For a given advice of type `Advice m a b`, we call `a → m b` the *advised type* of the advice.

Aspect. An aspect is a first-class value binding together a pointcut and an advice. Supporting first-class aspects is important: it makes it possible to support aspect factories, separate creation and deployment/undeployment of aspects, exporting opaque, self-contained aspects as single units, etc. We introduce a data definition for aspects, parameterized by a monad `m` (which has to be the same in the pointcut and advice):

[2] Haskell has a mechanism to introspect types called `Typeable`, but it is limited only to monomorphic types. `PolyTypeable` is an extension that supports polymorphic types and thus can be defined for any type.

[3] For this notion of function equality, we use the `StableNames` API, which relies on pointer comparison. See Section 7.1 for discussion on the issues of this approach.

```
data Aspect m a b c d = Aspect (PC m a b) (Advice m c d)
```

We defer the detailed definition of `Aspect` with its type class constraints to Section 3.2, when we address the issue of safe pointcut/advice binding.

2.2 Aspect Deployment

The list of aspects that are deployed at a given point in time is known as the *aspect environment*. To be able to define an heterogenous list of aspects, we use an existentially-quantified data `EAspect` that hides the type parameters of `Aspect`:[4]

```
data EAspect m = ∀ a b c d. EAspect (Aspect m a b c d)

type AspectEnv m = [EAspect m]
```

This environment can be either fixed initially and used globally [19], as in AspectJ, or it can be dynamic, as in AspectScheme [9]. Different scoping strategies are possible when dealing with dynamic deployment [31]. Since we are in a monadic setting, we can pass the aspect environment implicitly using a monad. An open function application can then trigger the set of currently-deployed aspects by retrieving these aspects from the underlying monad.

There are a number of design options for the aspect environment, depending on the kind of aspect deployment that is desired. Following the `Reader` monad, we can provide a fixed aspect environment, and add the ability to deploy an aspect for the dynamic extent of an expression, similarly to the `local` method of the `Reader` monad. We can also adopt a `State`-like monad, in order to support dynamic aspect deployment and undeployment with global scope. In this paper, without loss of generality, we go for the latter.

Because we are interested in using arbitrary computational effects in programs, we define the aspect environment through a *monad transformer*, which allows the programmer to construct a monadic stack of effects [17]. A monad transformer is a type constructor that is applied to an underlying monad to construct a new monad enhanced with the effect introduced by the transformer, while retaining access to all the underlying effects. The `AOT` monad transformer is defined as follows:

```
data AOT m a = AOT {run :: AspectEnv (AOT m) →
                          m (a, AspectEnv (AOT m))}
```

Similar to the state transformer, we use a **`data`** declaration to define the type `AOT`. This type wraps a `run` function, which takes an initial aspect environment and returns a computation in the underlying monad `m` with a value of type `a`, and a potentially modified aspect environment.

The monadic `bind` and `return` functions of the composed `AOT m` monad are the same as in the state monad transformer. Note that the aspect environment is bound to the same monad `AOT m`. This provides aspects with access to open applications[5].

We now define the functions for dynamic deployment, which simply add and remove an aspect from the aspect environment (note the use of `$` to avoid extra parentheses):

```
deploy, undeploy :: EAspect (AOT m) → AOT m ()
deploy asp   = AOT $ \asps → return ((), asp:asps)
undeploy asp = AOT $ \asps → return ((), deleteAsp asp asps)
```

To extract the computation of the underlying monad from an `AOT` computation we define the `runAOT` function, with type `Monad m ⇒ AOT m a → m a` (similar to `evalStateT` in the state monad transformer), that runs a computation in an empty initial aspect environment. For instance, in the example of the `sqrt` function, we can define a `client` as follows:

```
client n = runIdentity (runAOT (program n))
```

2.3 Aspect Weaving

Aspect weaving is triggered through open applications, *i.e.* applications performed with the `#` operator, *e.g.* `f # x`.

Open applications. We introduce a type class `OpenApp` that declares the `#` operator. This makes it possible to overload `#` in certain contexts, and it can be used to declare constraints on monads to ensure that the operation is available in a given context.

```
class Monad m ⇒ OpenApp m where
  (#) :: (a → m b) → a → m b
```

The `#` operator takes a function of type `a → m b` and returns a (woven) function with the same type. Any monad composed with the `AOT` transformer has open application defined:

```
instance Monad m ⇒ OpenApp (AOT m) where
  f # a = AOT $ \asps →
            do woven_f ← weave f asps (newjp f a)
               run (woven_f a) asps
```

An open application results in the creation of a join point (`newjp`) that represents the application of `f` to `a`. The join point is then used to determine which aspects in the environment match, produce a new function that combines all the applicable advices, and apply that function to the original argument.

Weaving. The function to use at a given point is produced by the `weave` function, defined below:

```
weave :: Monad m ⇒ (a → AOT m b) → AspectEnv (AOT m)
           → JP (AOT m) a b → m (a → AOT m b)
weave f [] jp = return f
weave f env@(asp:asps) jp =
    case asp of EAspect (Aspect pc adv) →
      do (match,_) ← apply_pc pc jp env
         weave (if match
                  then apply_adv adv f
                  else f)
               asps jp
```

The `weave` function is defined recursively on the aspect environment. For each aspect, it applies the pointcut to the join point. It then uses either the partial application of the advice to `f` if the pointcut matches, or `f` otherwise, to keep on weaving on the rest of the aspect list. This definition is a direct adaptation of AspectScheme's weaving function [9].

Applying advice. As we have seen, the aspect environment has type `AspectEnv m`, meaning that the type of the advice function is hidden. Therefore, advice application requires *coercing* the advice to the proper type in order to apply it to the function of the join point:

```
apply_adv :: Advice m a b → t → t
apply_adv adv f = (unsafeCoerce adv) f
```

The operation `unsafeCoerce` of Haskell is (unsurprisingly) unsafe and can yield to segmentation faults or arbitrary results. To recover safety, we could insert a runtime type check with `compareType` just before the coercion. We instead make aspects type safe such that we can prove that the use of `unsafeCoerce` in `apply_adv` is *always* safe. The following section describes how we achieve type soundness of aspects; Section 4 formally proves it.

[4] Since existential quantification requires type parameters to be free of type class constraints, aspects with ad-hoc polymorphism have to be instantiated before deployment to statically solve each remaining type class constraint (see Section 7.2 for more details).

[5] We could have defined `AOT` using the state monad transformer. However this would cause conflicts with existing monad transformer libraries when composing several effects. For instance, deploying `AOT` on a monadic stack that already contains a state component would imply using explicit lifting. We integrate `AOT` as a monad transformer that implicitly lifts operations for standard effects such as state, errors, IO, etc.

3. Typing Aspects

Ensuring type soundness in the presence of aspects consists in ensuring that an advice is always applied at a join point of the proper type. Note that by "the type of the join point", we refer to the type of the function being applied at the considered join point.

3.1 Typing Pointcuts

The intermediary between a join point and an advice is the pointcut, whose proper typing is therefore crucial. The type of a pointcut as a predicate over join points does not convey any information about the types of join points it matches. To keep this information, we use *phantom type variables* `a` and `b` in the definition of `PC`:

```
data PC m a b = Monad m ⇒ PC (∀ a' b'. m (JP m a' b' → m Bool))
```

A phantom type variable is a type variable that is not used on the right hand-side of the data type definition. The use of phantom type variables to type embedded languages was first introduced by Leijen and Meijer to type an embedding of SQL in Haskell [16]; it makes it possible to "tag" extra type information on data. In our context, we use it to add the information about the type of the join points matched by a pointcut: `PC m a b` means that a pointcut can match applications of functions of type `a → m b`. We call this type the *matched type* of the pointcut. Pointcut designators are in charge of specifying the matched type of the pointcuts they produce.

Least general types. Because a pointcut potentially matches many join points of different types, the associated type must be a *more general type*. For instance, consider a pointcut that matches applications of functions of type `Int → m Int` and `Float → m Int`. Its matched type is the parametric type `a → m Int`. Note that this is in fact the *least general type* of both types.[6] Another more general candidate is `a → m b`, but the least general type conveys more precise information.

As a concrete example, below is the type signature of the `pcCall` pointcut designator:

```
pcCall :: Monad m ⇒ (a → m b) → PC m a b
```

Comparing types. The type signature of the `pcType` pointcut designator is the same as that of `pcCall`:

```
pcType :: Monad m ⇒ (a → m b) → PC m a b
```

However, suppose that `f` is a function of type `Int → m a`. We want the pointcut `(pcType f)` to match applications of functions of more specific types, such as `Int → m Int`. This means that `compareType` actually checks that the matched type of the pointcut is *more general* than the type of the join point.

Logical combinators. We use type constraints in order to properly specify the matched type of logical combinations of pointcuts. The intersection of two pointcuts matches join points that are most precisely described by the *principal unifier* of both matched types. Since Haskell supports this unification when the same type variable is used, we can simply define `pcAnd` as follows:

```
pcAnd :: Monad m ⇒ PC m a b → PC m a b → PC m a b
```

For instance, a control flow pointcut matches any type of join point, so its matched type is `a → m b`. Consequently, if `f` is of type `Int → m a`, the matched type of `pcAnd (pcCall f) (pcCflow g)` is `Int → m a`.

Dually, the union of two pointcuts relies on *anti-unification* [25, 26], that is, the computation of the least general type of two types. Haskell does not natively support anti-unification. We exploit the fact that multi-parameter type classes can be used to define relations over types, and develop a novel type class `LeastGen` (for *least general*) that can be used as a constraint to compute the least general type `t` of two types `t1` and `t2` (defined in Section 4):

```
pcOr :: (Monad m, LeastGen (a → b) (c → d) (e → f)) ⇒
        PC m a b → PC m c d → PC m e f
```

For instance, if `f` is of type `Int → m a` and `g` is of type `Int → m Float`, the matched type of `pcOr (pcCall f) (pcCall g)` is `Int → m a`.

The negation of a pointcut can match join points of any type because no assumption can be made on the matched join points:

```
pcNot :: Monad m ⇒ PC m a b → PC m a' b'
```

User-defined pointcut designators. The set of pointcut designators in our language is open. User-defined pointcut designators are however responsible for properly specifying their matched types. If the matched type is incorrect or too specific, soundness is lost.

A pointcut cannot make any type assumption about the type of the join point it receives as argument. The reason for this is again the homogeneity of the aspect environment: when deploying an aspect, the type of its pointcut is hidden. At runtime, then, a pointcut is expected to be applicable to any join point. The general approach to make a pointcut safe is therefore to perform a runtime type check, as was illustrated in the definition of `pcCall` and `pcType` in Section 2.1. However, certain pointcuts are meant to be conjuncted with others pointcuts that will first apply a sufficient type condition.

In order to support the definition of pointcuts that *require* join points to be of a given type, we provide the `RequirePC` type:

```
data RequirePC m a b = Monad m ⇒
                       RequirePC (∀ a' b'. m (JP m a' b' → m Bool))
```

The definition of `RequirePC` is similar to that of `PC`, with two important differences. First, the matched type of a `RequirePC` is interpreted as a type *requirement*. Second, a `RequirePC` is not a valid stand-alone pointcut: it has to be combined with a standard `PC` that enforces the proper type upfront. To safely achieve this, we overload `pcAnd`[7]:

```
pcAnd :: (Monad m, LessGen (a → b) (c → d)) ⇒
         PC m a b → RequirePC m c d → PC m a b
```

`pcAnd` yields a standard `PC` pointcut and checks that the matched type of the `PC` pointcut is *less general* than the type expected by the `RequirePC` pointcut. This is expressed using the constraint `LessGen`, which, as we will see in Section 4, is based on `LeastGen`.

To illustrate, let us define a poincut designator `pcArgGT` for specifying pointcuts that match when the argument at the join point is greater than a given `n` (of type `a` instance of the `Ord` type class):

```
pcArgGT :: (Monad m, Ord a) ⇒ a → RequirePC m a b
pcArgGT n = RequirePC $ return (\jp →
              return (unsafeCoerce (getJpArg jp) >= n))
```

The use of `unsafeCoerce` to coerce the join point argument to the type `a` forces us to declare the `Ord` constraint on `a` when typing the returned pointcut as `RequirePC m a b` (with a fresh type variable `b`). To get a proper pointcut, we use `pcAnd`, for instance to match all calls to `sqrtM` where the argument is greater than 10:

```
pcCall sqrtM `pcAnd` pcArgGT 10
```

The `pcAnd` combinator guarantees that a `pcArgGT` pointcut is always applied to a join point with an argument that is indeed of a proper type: no runtime type check is necessary within `pcArgGT`, because the coercion is always safe.

[6] The term *most specific generalization* is also valid, but we stick here to Plotkin's original terminology [25].

[7] The constraint is different from the previous constraint on `pcAnd`. This is possible thanks to the recent `ConstraintKinds` extension of `ghc`.

3.2 Typing Aspects

The main typing issue we have to address consists in ensuring that a pointcut/advice binding is type safe, so that the advice application does not fail. A first idea to ensure that the pointcut/advice binding is type safe is to require the matched type of the pointcut and the advised type of the advice to be the same (or rather, unifiable):

```
wrong!
data Aspect m a b = Aspect (PC m a b) (Advice m a b)
```

This approach can however yield unexpected behavior. Consider the following example:

```
idM x = return x

adv :: Monad m ⇒ Advice (Char → m Char)
adv proceed c = proceed (toUpper c)

program = do deploy (aspect (pcCall idM) adv)
             x ← idM # 'a'
             y ← idM # [True,False,True]
             return (x, y)
```

The matched type of the pointcut `pcCall idM` is `Monad m ⇒ a → m a`. With the above definition of `Aspect`, `program` passes the typechecker because it is possible to unify `a` and `Char` to `Char`. However, when evaluated, the behavior of `program` is undefined because the advice is unsafely applied with an argument of type `[Bool]`, for which `toUpper` is undefined.

The problem is that during typechecking, the matched type of the pointcut and the advised type of the advice can be unified. Because unification is symmetric, this succeeds even if the advised type is more specific than the matched type. In order to address this, we again use the type class `LessGen` to ensure that the matched type is less general than the advice type:

```
data Aspect m a b c d = (Monad m, LessGen (a → m b) (c → m d))
                      ⇒  Aspect (PC m a b) (Advice m c d)
```

This constraint ensures that pointcut/advice bindings are type safe: the coercion performed in `apply_adv` always succeeds. We formally prove this in the following section.

4. Typing Aspects, Formally

We now formally prove the safety of our approach. We start briefly summarizing the notion of type substitutions and the *is less general* relation between types. Note that we do not consider type class constraints in the definition. Then we describe a novel anti-unification algorithm implemented with type classes, on which the type classes `LessGen` and `LeastGen` are based. We finally prove pointcut and aspect safety, and state our main safety theorem.

4.1 Type Substitutions

In this section we summarize the definition of type substitutions, which form the basis of our argument for safety. We consider a typing environment $\Gamma = (x_i : T_i)_{i \in \mathbb{N}}$ that binds variables to types.

Definition 1 (Type Substitution, from [24])**.** A type substitution σ is a finite mapping from type variables to types. It is denoted $[X_i \mapsto T_i]_{i \in \mathbb{N}}$, where $dom(\sigma)$ and $range(\sigma)$ are the sets of types appearing in the left-hand and right-hand sides of the mapping, respectively. It is possible for type variables to appear in $range(\sigma)$.

Substitutions are always applied simultaneously on a type. If σ and γ are substitutions, and T is a type, then $\sigma \circ \gamma$ is the composed substitution, where $(\sigma \circ \gamma)T = \sigma(\gamma T)$. Application of substitution on a type is defined inductively on the structure of the type.

Substitution is extended pointwise for typing environments in the following way: $\sigma(x_i : T_i)_{i \in \mathbb{N}} = (x_i : \sigma T_i)_{i \in \mathbb{N}}$. Also, applying a substitution to a term t means to apply the substitution to all type annotations appearing in t.

```
1  class LeastGen' a b c σ_in σ_out | a b c σ_in → σ_out
2
3  Inductive case: The two type constructors match,
4  recursively compute the substitution for type arguments a_i, b_i.
5  instance (LeastGen' a_1 b_1 c_1 σ_0 σ_1, ...,
6            LeastGen' a_n b_n c_n σ_{n-1} σ_n,
7            T c_1 ... c_n ~ c)
8         ⇒ LeastGen' (T a_1 ... a_n) (T b_1 ... b_n) c σ_0 σ_n
9
10 Default case: The two type constructors don't match, c has to be a variable,
11 either unify c with c' if c' ↦ (a, b) or extend the substitution with c ↦ (a, b)
12 instance (Analyze c (TVar c),
13           MapsTo σ_in c' (a,b),
14           VarCase c' (a,b) c σ_in σ_out)
15        ⇒ LeastGen' a b c σ_in σ_out
```

Figure 1. `LeastGen'`

Definition 2 (Less General Type)**.** We say type T_1 *is less general* than type T_2, denoted $T_1 \preceq T_2$, if there exists a substitution σ such that $\sigma T_2 = T_1$. Observe that $\preceq$ defines a partial order on types (modulo α-renaming).

Definition 3 (Least General Type)**.** Given types T_1 and T_2, we say type T is the *least general type* iff T is the supremum of T_1 and T_2 with respect to $\preceq$.

4.2 Statically Computing Least General Types

In an aspect declaration, we statically check the type of the pointcut and the type of the advice to ensure a safe binding. To do this we encode an anti-unification algorithm at the type level, exploiting the type class mechanism of Haskell. A multi-parameter type class `R` $t_1 \ldots t_n$ can be seen as a *relation* `R` on types $t_1 \ldots t_n$, and **`instance`** declarations as ways to (inductively) define this relation, in a manner very similar to logic programming.

The type classes `LessGen` and `LeastGen` used in Section 3 are defined as particular cases of the more general type class `LeastGen'`, shown in Figure 1. This class is defined in line 1 and is parameterized by types a, b, c, σ_{in} and σ_{out}. σ_{in} and σ_{out} denote substitutions encoded at the type level as a list of mappings from type variables to *pairs* of types. We use pairs of types in substitutions because we have to simultaneously compute substitutions from c to a and from c to b[8]. To be concise, lines 5-8 present a single definition parametrized by the type constructor arity but in practice, there needs to be a different instance declaration for each type constructor arity.

Proposition 1. If `LeastGen'` $a\ b\ c\ \sigma_{in}\ \sigma_{out}$ holds, then the substitution σ_{out} extends σ_{in} and $\sigma_{out} c = (a, b)$.

Proof. By induction on the type representation of a and b.

A type can either be a type variable, represented as `TVar a`, or an n-ary type constructor `T` applied to n type arguments[9]. The rule to be applied depends on whether the type constructors of a and b are the same or not.

(*i*) If the constructors are the same, the rule defined in lines 5-8 computes `(T` c_1 ... c_n`)` using the induction hypothesis that $\sigma_i c_i = (a_i, b_i)$, for $i = 1 \ldots n$. The component-wise application of constraints is done from left to right, starting from substitution σ_0 and

[8] The $a\ b\ c\ \sigma_{in} \rightarrow \sigma_{out}$ expression means that σ_{out} is functionally dependent on the other parameters. Functional dependencies were proposed by Jones [13] as a mechanism to more precisely control type inference in Haskell. An expression `c e | c → e` means that fixing the type `c` should fix the type `e`.

[9] We use the `Analyze` type class from `PolyTypeable` to get a type representation at the type level. For simplicity we omit the rules for analyzing type representations.

extending it to the resulting substitution σ_n. The type equality constraint `(T` c_1 `...)` $\sim c$ checks that `c` is unifiable with `(T` c_1 `...)` and, if so, unifies them. Then, we can check that $\sigma_n c = (a, b)$.

(*ii*) If the type constructors are not the same the only possible generalization is a type variable. In the rule defined in lines 12-15 the goal is to extend σ_{in} with the mapping $c \mapsto (a, b)$ such that $\sigma_{out} c = (a, b)$, while preserving the injectivity of the substitution (see next proposition). □

Proposition 2. If σ_{in} is an injective function, and `LeastGen'` $a\ b\ c\ \sigma_{in}\ \sigma_{out}$ holds, then σ_{out} is an injective function.

Proof. By construction `LeastGen'` introduces a binding from a fresh type variable to (a, b), in the rule defined in lines 12-15, only if there is no type variable already mapping to (a, b)—in which case σ_{in} is not modified.

To do this, we first check that c is actually a type variable `(TVar` c`)` by checking its representation using `Analyze`. Then in relation `MapsTo` we bind c' to the (possibly inexistent) type variable that maps to (a, b) in σ_{in}. In case there is no such mapping c' is `None`.

Finally, relation `VarCase` binds σ_{out} to σ_{in} extended with $\{c \mapsto (a, b)\}$ in case c' is `None`, otherwise $\sigma_{out} = \sigma_{in}$. It then unifies c with c'. In all cases c is bound to the variable that maps to (a, b) in σ_{out}, because it was either unified in rule `MapsTo` or in rule `VarCase`.

The hypothesis that σ_{in} is injective ensures that any preexisting mapping is unique. □

Proposition 3. If σ_{in} is an injective function, and `LeastGen'` $a\ b\ c\ \sigma_{in}\ \sigma_{out}$ holds, then c is the least general type of a and b.

Proof. By induction on the type representation of a and b.

(*i*) If the type constructors are different the only generalization possible is a type variable c.

(*ii*) If the type constructors are the same, then $a = Ta_1 \dots a_n$ and $b = Tb_1 \dots b_n$. By Proposition 1 , $c = Tc_1 \dots c_n$ generalizes a and b with the substitution σ_{out}. By induction hypothesis c_i is the least general type of (a_i, b_i).

Now consider a type d that also generalizes a and b, *i.e.* $a \preceq d$ and $b \preceq d$, with associated substitution α. We prove c is less general than d by constructing a substitution τ such that $\tau d = c$.

Again, there are two cases, either d is a type variable, in which case we set $\tau = \{d \mapsto c\}$, or it has the same outermost type constructor, *i.e.* $d = Td_1 \dots d_n$. Thus $a_i \preceq d_i$ and $b_i \preceq d_i$; and since c_i is the least general type of a_i and b_i, there exists a substitution τ_i such that $\tau_i d_i = c_i$, for $i = 1 \dots n$.

Now consider a type variable $x \in dom(\tau_i) \cap dom(\tau_j)$. By definition of α, we know that $\sigma_{out}(\tau_i(x)) = \alpha(x)$ and $\sigma_{out}(\tau_j(x)) = \alpha(x)$. Because σ_{out} is injective (by Proposition 2), we deduce that $\tau_i(x) = \tau_j(x)$ so there are no conflicting mappings between τ_i and τ_j, for any i and j. Thus we can define $\tau = \bigcup \tau_i$ and check that $\tau d = c$. □

Definition 4 (`LeastGen` type class)**.** To compute the least general type `c` for `a` and `b`, we define:

`LeastGen a b c` $\triangleq$ `LeastGen' a b c` $\sigma_{empty}\ \sigma_{out}$, where σ_{empty} is the empty substitution and σ_{out} is the resulting substitution.

Definition 5 (`LessGen` type class)**.** To establish that type `a` is less general than type `b`, we define:

`LessGen a b` $\triangleq$ `LeastGen a b b`

4.3 Pointcut Safety

We now establish the safety of pointcuts with relation to join points.

Definition 6 (Pointcut match)**.** We define the relation `matches(pc,jp)`, which holds iff applying pointcut `pc` to join point `jp` in the context of a monad `m` yields a computation `m True`.

Now we prove that the matched type of a given pointcut is more general than the join points matched by that pointcut.

Proposition 4. Given a join point term `jp` and a pointcut term `pc`, and type environment Γ, if
$\Gamma \vdash$ `pc: PC m a b`
$\Gamma \vdash$ `jp: JP m a' b'`
$\Gamma \vdash$ `matches(pc, jp)`
then `a' → m b'` $\preceq$ `a → m b`.

Proof. By induction on the matched type of the pointcut.

- Case `pcCall`: By construction the matched type of a `pcCall f` pointcut is the type of `f`. Such a pointcut matches a join point with function `g` if and only if: `f` is equal to `g`, and the type of `f` is less general than the type of `g`. (On both `pcCall` and `pcType` this type comparison is performed by `compareType` on the type representations of its arguments.)
- Case `pcType`: By construction the matched type of a `pcType f` pointcut is the type of `f`. Such a pointcut only matches a join point with function `g` whose type is less general than the matched type.
- Case `pcAnd` on `PC PC`: Consider `pc1 `pcAnd` pc2`. The matched type of the combined pointcut is the *principal unifier* of the matched types of the arguments—which represents the intersection of the two sets of join points. The property holds by induction hypothesis on `pc1` and `pc2`.
- Case `pcAnd` on `PC RequirePC`: Consider `pc1 `pcAnd` pc2`. The matched type of the combined pointcut is the type of `pc1` and it is checked that the type required by `pc2` is *more general* so the application of `pc2` will not yield an error. The property holds by induction hypothesis on `pc1`.
- Case `pcOr`: Consider `pc1 `pcOr`pc2`. The matched type of the combined pointcut is the *least general type* of the matched types of the argument, computed by the `LeastGen` constraint—which represents the union of the two sets of join points. The property holds by induction hypothesis on `pc1` and `pc2`.
- Case `pcNot`: The matched type of a pointcut constructed with `pcNot` is a fresh type variable, which by definition is more general than the type of any join point.
- User-defined pointcuts must maintain this property, otherwise safety is lost.

□

4.4 Advice Type Safety

If an aspect is well-typed, the advice is more general than the matched type of the pointcut:

Proposition 5. Given a pointcut term `pc`, an advice term `adv`, and a type environment Γ, if
$\Gamma \vdash$ `pc: PC m a b`
$\Gamma \vdash$ `adv: Advice m c d`
$\Gamma \vdash$ `(aspect pc adv): Aspect m a b c d`
then `a → m b` $\preceq$ `c → m d`.

Proof. Using the definition of `Aspect` (Section 3.2) and because $\Gamma \vdash$ `(aspect pc adv): Aspect m a b c d`, we know that the constraint `LessGen` is satisfied, so by Definitions 4 and 5, and Proposition 1, `a → m b` $\preceq$ `c → m d`. □

4.5 Safe Aspects

We now show that if an aspect is well-typed, the advice is more general than the advised join point:

Theorem 1 (Safe Aspects)**.** *Given the terms* `jp`*,* `pc` *and* `adv` *representing a join point, a pointcut and an advice respectively, given a type environment* Γ*, if*
$\Gamma \vdash$ `pc: PC m a b`
$\Gamma \vdash$ `adv: Advice m c d`

```
module Fib (fib, pcFib) where
import AOP

fibBase n = return 1

pcFib = pcCall fibBase `pcAnd` pcArgGT 2

fibAdv proceed n = do f1 ← fibBase # (n-1)
                      f2 ← fibBase # (n-2)
                      return (f1 + f2)

fib :: Monad m ⇒ m (Int → m Int)
fib = do { deploy (aspect pcFib fibAdv); return $ fibBase # }
```

Figure 2. Fibonacci module.

$\Gamma \vdash$ `(aspect pc adv): Aspect m a b c d`
and
$\Gamma \vdash$ `jp: JP m a' b'`
$\Gamma \vdash$ `matches(pc, jp)`
then `a' → m b'` $\preceq$ `c → m d`.

Proof. By Proposition 4 and 5 and the transitivity of $\preceq$. □

Corollary 1 (Safe Advice Application)**.** The coercion of the advice in `apply_adv` is safe.

Proof. Recall `apply_adv` (Section 2.3):

```
apply_adv :: Advice m a b → t → t
apply_adv adv f = (unsafeCoerce adv) f
```

By construction, `apply_adv` is used only with a function `f` that comes from a join point that is matched by a pointcut associated to `adv`. Using Theorem 1, we know that the join point has type `JP m a' b'` and that `a' → m b'` $\preceq$ `a → m b`. We note σ the associated substitution. Then, by compatibility of substitutions with the typing judgement [24], we deduce $\sigma\Gamma \vdash \sigma$`adv: Advice m a' b'`. Therefore `(unsafeCoerce adv)` corresponds exactly to σ`adv`, and is safe. □

5. Type-Based Reasoning About Aspects

This section illustrates how we can exploit the monadic embedding for type-based reasoning about aspects, regarding both control flow properties and computational effects. In essence, this section shows how the approach of EffectiveAdvice [23] can be used in the context of pointcut/advice AOP, or dually, how Open Modules [1] can be extended with effects.

5.1 A Simple Example

We first describe a simple example that serves as the starting point. Figure 2 describes a Fibonacci module, following the canonical example of Open Modules. The module uses an internal aspect to implement the recursive definition of Fibonacci: the base function, `fibBase` simply implements the base case, and the `fibAdv` advice implements recursion when the pointcut `pcFib` matches. Note that `pcFib` uses the user-defined poincut `pcArgGT` (defined in Section 3.1) to check that the call to `fibBase` is done with an argument greater than 2. The `fib` function is defined by first deploying the internal aspect, and then partially applying `#` to `fibBase`. This transparently ensures that an application of `fib` is open. The `fib` function is exported, together with the `pcFib` pointcut, which can be used by an external module to advise applications of the internal `fibBase` function. Figure 3 presents another Haskell module that provides a more efficient implementation of `fib` by using a memoization advice. To benefit from memoization, a client only has to import `fib` from the `MemoizedFib` module instead of directly from the `Fib` module.

Note that, if we consider that the aspect language only supports the `pcCall` pointcut designator, this implementation actually represents an open module proper. Preserving the properties of open modules, in particular protecting from external advising of internal functions, in presence of arbitrary quantification (*e.g.* `pcType`, or an always-matching pointcut) is left for future work. Importantly, just like Open Modules, the approach described here does not ensure anything about the advice beyond type safety. In particular, it is possible to create an aspect that incorrectly calls `proceed` several times, or an aspect that has undesired computational effects. The type system can assist us in expressing and enforcing specific interference properties.

```
module MemoizedFib (fib) where
import qualified Fib
import AOP

memo proceed n =
 do m ← get
    if member n m then return (m ! n)
    else do { y  ← proceed n ; m' ← get; put (insert n y m');
              return y }

fib = do { deploy (aspect Fib.pcFib memo); Fib.fib }
```

Figure 3. Memoized Fibonacci module.

5.2 Protected Pointcuts

In order to extend Open Modules with effect-related enforcement, we introduce the notion of *protected pointcuts*, which are pointcuts enriched with restrictions on the effects that associated advice can exhibit. Simply put, a protected pointcut embeds a *typed combinator* that is applied to the advice in order to build an aspect. If the advice does not respect the (type) restrictions expressed by the combinator, the aspect creation expression simply does not typecheck and hence the aspect cannot be built.

A typed combinator is any function that can produce an advice:

```
type Combinator t m a b = Monad m ⇒ t → Advice m a b
```

The `protectPC` function packs together a pointcut and a combinator:

```
protectPC :: (Monad m, LessGen (a → m b) (c → m d)) ⇒
   PC m a b → Combinator t m c d → ProtectedPC m a b t c d
```

A protected pointcut, of type `ProtectedPC`, cannot be used with the standard aspect creation function `aspect`. The following `pAspect` function is the only way to get an aspect from a protected pointcut (the constructor `PPC` is not exposed):

```
pAspect :: Monad m ⇒ ProtectedPC m a b t c d → t
                   → Aspect m a b c d
pAspect (PPC pc comb) adv = aspect pc (comb adv)
```

The key point here is that when building an aspect using a protected pointcut, the combinator `comb` is applied to the advice `adv`.

We now show how to exploit this extension of Open Modules to control both control flow and computational effects, using the proper type combinators.

5.3 Control Flow Properties

Rinard *et al.* present a classification of advice in four categories depending on how they affect the control flow of programs [27]:

- **Combination:** The advice can call `proceed` any number of times.
- **Replacement:** There are no calls to `proceed` in the advice.
- **Augmentation:** The advice calls `proceed` exactly once, and does not modify the arguments to or the return value of `proceed`.
- **Narrowing:** The advice calls `proceed` at most once, and does not modify the arguments to or the return value of `proceed`.

```
type Narrow m a b c = Monad m ⇒
          (a → m Bool, Augment m a b c, Replace m a b)
narrow :: Monad m ⇒ Narrow m a b c → Advice m a b
narrow (p, aug, rep) proceed x =
  do b ← p x
     if b then replace rep proceed x
          else augment aug proceed x
```

Figure 4. Narrowing advice combinator (adapted from [23]).

```
fibErr :: (..., MonadError String (t m)) ⇒
          NIAOT t m (Int → NIAOT t m Int)
fibErr = do deploy (niAspect pcFib fibAdv)
            return errorFib
       where errorFib n = if n < 0
                          then throwError "Not defined"
                          else fibBase # n
```

Figure 5. Fibonacci with error.

Memoization is a typical example of a narrowing advice: the combination of a replacement advice ("return memoized value without proceeding") and an augmentation advice ("proceed and memoize return value"), where the choice between both is driven by a runtime predicate ("is there a memoized value for this argument?").

Oliveira *et al.* [23] show a type-based enforcement of these categories, through advice combinators. Figure 4 shows the definition of the `narrow` combinator, which takes a triple consisting of the predicate, the augmentation advice and the replacement advice, and builds a proper advice with the narrowing logic. For brevity, we do not detail all combinators and advice types, taken from [23]. These combinators fit the general `Combinator` type we described in Section 5.2, and can therefore be embedded in protected pointcuts. It is now straightforward for the `Fib` module to expose a protected pointcut that restricts valid advice to narrowing advice:

```
module Fib (fib, ppcFib) where
...
ppcFib = protectPC pcFib narrow
```

The protected pointcut `ppcFib` embeds the `narrow` type combinator. Therefore, only advice that can be statically typed as narrowing advice can be bound to that pointcut.

5.4 Effect Interference

The typed monadic embedding of aspects also allows to reason about computational effects. For instance, consider a modification of the Fibonacci module where the `fibErr` function can throw an error message when called with a negative integer (Figure 5). In that situation, it is interesting to ensure that the advice bound to the exposed pointcut cannot throw or catch in the same Error monad.

This can be done as in EffectiveAdvice [23], by enforcing advices to be parametric with respect to the monad used by the base computation. To distinguish between base and aspect parts of the monadic stack, we first have to introduce a modified `AOT` monad transformer that manages a splitting of the monadic stack into a monad transformer `t` that collects effects available to aspect computations and a monad `m` that collects effects available to the base computation. Thus, we define the data type `NIAOT` as follows (`NI` stands for non-interference):

```
data NIAOT t m a = NIAOT {runNI :: AspectEnv (NIAOT t m)
                                → t m (a, AspectEnv (NIAOT t m)) }
```

and extend other definitions (`weave`, `deploy`, ...) accordingly.

Effect interference and pointcuts. The novelty compare to EffectiveAdvice is that we also have to deal with interferences for pointcuts. But to allow effect-based reasoning on pointcuts, we need to distinguish between the monad used by the base computation and the monad used by pointcuts. Indeed, in the interpretation of the type `PC m a b`, `m` stands for both monads, which forbids to reason separately about them. To address this issue, we need to interpret `PC m a b` differently, by saying that the matched type is `a → b` instead of `a → m b`. In this way, the monad for the base computation (which is implicitly bound by `b`) does not have to be `m` at the time the pointcut is defined. To accommodate this new interpretation with the rest of the code, very little changes have to be made[10]. The main changes are in the type of `pcCall`, `pcType` and in the definition of `Aspect`

```
pcCall, pcType :: Monad m ⇒ (a → b) → PC m a b

data Aspect m a b c d = (Monad m, LessGen (a → b) (c → m d)) ⇒
                        Aspect (PC m a b) (Advice m c d)
```

Note how the definition of `Aspect` forces the monad of the pointcut computation to be unified with that of the advice, and with that of the base code. The result of Section 4 can straightforwardly be rephrased with these new definitions.

Typing non-interfering poincuts and advices. It now becomes possible to restrict the type of pointcuts and advices by using rank-2 types. The following type synonyms guarantee that an aspect of type `NIPC t a b` only uses effects available in `t` (and similarly for `NIAdvice t a b`).

```
type NIPC t a b = ∀ m. (Monad (t m), Monad m, ...) ⇒
                  PC (NIAOT t m) a b
type NIAdvice t a b = ∀ m. (Monad (t m), Monad m, ...) ⇒
                      Advice (NIAOT t m) a b
```

By universally quantifying over the type `m` of the effects used in the base computation, these types enforce, through the properties of parametricity, that pointcuts (or advices) cannot refer to specific effects in the base program.

We can define aspect construction functions that enforce different (non-)interference patterns, such as non-interfering pointcut `NIPC` with unrestricted advice `Advice`, unrestricted pointcut `PC` with non-interfering advice `NIAdvice`, etc. Symmetrically, we can check that a part of the base code cannot interfere with effects available to aspects by using the type synonym `NIBase`, which universally quantifies over the type `t` of effects available to the advice:

```
type NIBase m a b = ∀ t. (Monad (t m), MonadTrans t, ...) ⇒
                    a → NIAOT t m b
```

Coming back to Open Modules and protected pointcuts, in order to enforce non-interfering advice, we need to define a typed combinator that requires an advice of type `NIAdvice`:

```
niAdvice :: (Monad (t m), Monad m) ⇒
            NIAdvice t a b → Advice (NIAOT t m) a b
niAdvice adv = adv
```

The `niAdvice` combinator is computationally the identity function, but does impose a type requirement on its argument. Using this combinator, the `Fib` module can expose a protected pointcut that enforces non-interference with base effects:

```
module Fib (fib, ppcFib) where
...
ppcFib = protectPC pcFib niAdvice
```

EffectiveAdvice is restricted with respect to AOP in that it does not support quantification at all [23]. We have shown that in our embedding, protected pointcuts can be used to extend EffectiveAdvice to support quantification, thereby combining it with Open Modules.

[10] The implementation available online uses this interpretation of `PC m a b`.

```
type Level = Int
data ELT m a = ELT {run :: Level -> m (a, Level)}
 primitive operations
inc  = ELT $ \l -> return ((), l + 1)
dec  = ELT $ \l -> return ((), l - 1)
at l = ELT $ \_ -> return ((), l)
 user visible operations
current = ELT $ \l -> return (l, l)
up c    = do {inc; result <- c; dec; return result}
down c  = do {dec; result <- c; inc; return result}
lambda_at f l = \arg -> do
{n <- current; at l; result <- f arg; at n; return result}
```

Figure 6. Execution levels monad transformer and level-shifting operations

6. Language Extensions

The typed monadic embedding of aspects supports modular extensions of the aspect language. More precisely, we can modularly implement new semantics for aspect scoping and weaving. We briefly discuss in this section three possible extensions (available in the online distribution): i) aspect weaving with execution levels [32]; ii) secure weaving in which a set of join points can be hidden from advising; iii) privileged aspects that can see hidden join points. At the end of this section, we discuss how the types allow us to reason about these extensions.

6.1 Execution Levels

Execution levels avoid unwanted *computational interference* between aspects, *i.e.* when an aspect execution produces join points that are visible to others, including itself [32]. Execution levels give structure to execution by establishing a tower in which the flow of control navigates. Aspects are deployed at a given level and can only affect the execution of the underlying level. The execution of an aspect (both pointcuts and advices) is therefore not visible to itself and to other aspects deployed at the same level, only to aspects standing one level above. The original computation triggered with `proceed` is always executed at the level at which the join point was emitted. If needed, the programmer can use level-shifting operators to move execution up and down in the tower.

The monadic semantics of execution levels are implemented in the `ELT` monad transformer (Figure 6). The `Level` type synonym represents the level of execution as an integer. `ELT` wraps a `run` function that takes an initial level and returns a computation in the underlying monad `m`, with a value of type `a` and a potentially-modified level. As in the `AOT` transformer, the monadic `bind` and `return` functions are the same as in the state monad transformer. The private operations `inc`, `dec`, and `at` are used to define `current`, `up`, `down`, and `lambda_at`. In addition to level shifting with `up` and `down`, `current` reifies the current level, and `lambda_at` creates a *level-capturing function* bound at level `l`. When such a function is applied, execution jumps to level `l` and then goes back to the level prior to the application [32].

The semantics of execution levels can be embedded in the definition of aspects themselves, by transforming the pointcut and advice of an aspect at deployment time[11]. This is done through the two functions, `pcEL` and `advEL` (Figure 7). `pcEL` first ensures that the current execution level `lapp` matches `ldep`, the level at which the aspect is deployed. If so it then runs the pointcut one level above. Similarly, `advEL` ensures that the advice is run one level above, with a `proceed` function that captures the deployment level.

[11] For simplicity, in Section 2.2 we only described the default semantics of aspect deployment; aspect (un)deployement is actually defined using overloaded (un)`deployInEnv` functions.

```
deployInEnv (Aspect (pc::PC (AOT (ELT m)) tpc) adv) aenv =
  let pcEL ldep = (PC $ return (\jp → do
                    lapp ← current
                    if lapp == ldep
                    then up $ runPC pc jp
                    else return False)) :: PC (AOT (ELT m)) tpc
      advEL ldep proceed arg =
                  up $ adv (lambda_at proceed ldep) arg
  in do l ← current
        return EAspect (Aspect (pcEL l) (advEL l)) : aenv
```

Figure 7. Redefining aspect deployment for execution levels semantics. An aspect is made level-aware by transforming its pointcut and advice.

6.2 Secure Weaving

For security reasons it can be interesting to protect certain join points from being advised. To support such a secure weaving, we define a new monad transformer `AOT_s`, which embeds an (existentially quantified) pointcut that specifies the hidden join points, and we modify the weaving process accordingly (not shown here).

```
data EPC m = forall a b. EPC (PC m a b)

data AOT_s m a = AOT_s { runAOT_s ::
        AspectEnv (AOT_s m) → EPC (AOT_s m) →
        m (a, (AspectEnv (AOT_s m), EPC (AOT_s m)))}
```

This can be particularly useful when used with the `cflow` pointcut (described below) to protect the computation that occurs in the control flow of critical function applications. For instance, we can ensure that the whole control flow of function `f` is protected from advising during the execution of program `p`:

```
runAOT_s (EPC (pcCflow f)) p
```

`cflow` ***pointcut.*** The `pcCflow` pointcut is defined using a (join point) stack monad and one aspect that matches every join point and stores it in the stack; then the `pcCflow` pointcut is just a test on this stack. Using effect non-interference enforcement (Section 5.4), we can guarantee that this stack is private to `pcCflow`. Alternative optimizations can be defined, for example putting in the stack only relevant join points, or a per-flow deployment that allows to use a boolean instead of a stack.

6.3 Privileged Aspects

Hiding some join points to *all* aspects may be too restrictive. For instance, certain "system" aspects like access control should be treated as privileged and view all join points. Another example is the aspect in charge of maintaining the join point stack for the sake of control flow reasoning (used by `pcCflow`). In such cases, it is important to be able to define a set of privileged aspects, which can advise all join points, even those that are normally hidden. The implementation of a privileged aspects list is a straightforward extension to the secure weaving mechanism described above.

6.4 Reasoning about Language Extensions

The above extensions can be implemented in an untyped language such as LAScheme [32]. However, it is not possible in an untyped setting to statically reason about effects provided by a language extension or enforce that a piece of code is used with a particular weaving semantics.

Non-interference with the effects of language extensions. We can combine the monadic interpretation of execution levels with the management of effect interference (Section 5.4) in order to reason about level-shifting operations performed by base and aspect computations. For instance, it becomes possible to prevent aspect

and/or base computation to use effects provided by the ELT monad transformer, thus ensuring that the default semantics of execution levels is preserved (and therefore that the program is free of aspect loops [32]). If more advanced use of execution levels is required, this contraint can be explicitly relaxed in the AOT monad transformer, thus stressing in the type that it is the responsability of the programmer to avoid infinite regression.

Enforcing a particular weaving semantics through typing. The type system makes it possible to specify functions that can be woven, but only within a specific aspect monad. For instance, suppose that we want to define a `critical` computation, which should only be run with secure weaving for access control. The computation must therefore be run within the `AOT_s` monad transformer with a given pointcut `pc_ac` (`ac` stands for access control).

To enforce the use of `AOT_s` with a specific pointcut value would require the use of a dependent type, which is not possible in Haskell. This said, we can use the `newtype` data constructor together with its ability to derive automatically type class instances, to define a new type `AOT_ac` that encapsulates the `AOT_s` monad transformer and forces it to be run with the `pc_ac` pointcut:

```
newtype AOT_ac m a = AOT_ac (AOT_s m a)
        deriving (Monad, OpenApp, ...)

runSafe (AOT_ac c) = runAOT_s (EPC pc_ac) c
```

Therefore, we can export the `critical` computation by typing it appropriately:

```
critical :: Monad m ⇒ AOT_ac m a
```

Because the `AOT_ac` constructor is hidden in a module, the only way to run such a computation typed as `AOT_ac` is to use `runSafe`. The `critical` computation is therefore only advisable with secure weaving for access control. Type based reasoning about aspect language extensions is, to the best of our knowledge, a novel contribution of this work.

7. Discussion

We now discuss a number of issues related to our approach: how to define a proper notion of function equality, how to deal with overloaded functions, how to enhance the handling of the monadic stack, and finally, we analyze the issue of obliviousness.

7.1 Supporting Equality on Functions

Pointcuts quantify about join points, and a major element of the join point is the function being applied. The `pcType` designator relies on type comparison, implemented using the `PolyTypeable` type class in order to obtain representations for polymorphic types. The `pcCall` is more problematic, as it relies on function equality, but Haskell does not provide an operator like `eq?` in Scheme.

A first workaround is to use the `StableNames` API that allows to compare functions using pointer equality. Unfortunately, this notion of equality is fragile. `StableNames` equality is safe in the sense that it does not equate two functions that are not the same, but two functions that are equal can be seen as different.

The problem becomes even more systematic when it comes to bounded polymorphism. Indeed, each time a function with constraints is used, a new closure is created by passing the current method dictionary of type class instances. Even with optimized compilation (*e.g.* `ghc -O`), this (duplicated) closure creation is unavoidable and so `StableNames` will consider different any two constrained functions, even if the passed dictionary is the same.

To overcome this issue, we have overloaded our equality on functions with a special case for functions that have been explicitly tagged with a unique identifier at creation (using `Data.Unique`). This allows to have a robust notion of function equality but it has to be used explicitly at each function definition site.

7.2 Advising Overloaded Functions

From a programmer point of view, it can be interesting to advise an overloaded function (that is, the application of all the possible implementations) with a single aspect. However, deploying aspects in the general case of bounded polymorphism is problematic because of the resolution of class constraints. Recall that in order to be able to type the aspect environment, we existentially hide the matched and advised types of an aspect. This means that all type class constraints must be solved statically at the point an aspect is deployed. If the matched and advised types are both bounded polymorphic types, type inference cannot gather enough information to statically solve the constraints. So advising all possible implementations requires repeating deployment of the same aspect with different type annotations, one for each instance of the involved type classes.

To alleviate this problem, we developed a macro using Template-Haskell [28]. The macro extracts all the constrained variables in the matched type of the pointcut, and generates an annotated deployment for every possible combination of instances that satisfy all constraints. In order to retain safety, the advised type of an aspect must be less constrained than its matched type. This is statically enforced by the Haskell type system after macro expansion.

7.3 Obliviousness

The embedding of aspects we have presented thus far supports quantification through pointcuts, but is not oblivious: open applications are explicit in the code. A first way to introduce more obliviousness without requiring non-local macros or, equivalently, a preprocessor or ad hoc runtime semantics, is to use *partial applications* of #. For instance, the `sqrtM` function can be turned into an implicitly woven function by defining `sqrtM'= sqrtM #`. This approach was used in Figure 2 for the definition of `fib`. It can be sufficient in similar scenarios where quantification is under control. Otherwise, it can yield to issues in the definition of pointcuts that rely on function identity, because `sqrtM'` and `sqrtM` are different functions. Also, this approach is not entirely satisfactory with respect to obliviousness because it has to be applied specifically for each function.

In [21], De Meuter proposes to use the binder of a monad to redefine function application. His approach focuses on defining one monad per aspect, but can be generalized to a list of dynamically-deployed aspects as presented in Section 2.2. For this, we can redefine the monad transformer `AOT` to make all monadic applications open transparently:

```
instance Monad m ⇒ Monad (AOT m) where
    return a = AOT (\aenv → return (a, aenv))
    k >>= f  = do { x ← k; f # x }
```

This presentation improves obliviousness because any monadic application is now an open application, but it suffers from a major drawback: it breaks the *monadic laws*. Indeed, left identity and associativity can be invalidated, depending on the current list of deployed aspects. This is not surprising as AOP allows to redefine the behavior of a function and even to redefine the behavior of a function depending on its context of execution. Breaking monadic laws is not prohibited by Haskell, but it is very dangerous and fragile; for instance, some compilers exploit the laws to perform optimizations, so breaking them can yield to incorrect optimizations.

8. Related Work

The earliest connection between aspects and monads was established by De Meuter in 1997 [21]. In that work, he proposes to describe the weaving of a given aspect directly in the binder of a

monad. As we have just described above (Section 7.3), doing so breaks the monad laws, and is therefore undesirable.

Wand *et al.* [35] formalize pointcuts and advice and use monads to structure the denotational semantics; a monad is used to pass the join point stack and the store around evaluation steps. The specific flavor of AOP that is described is similar to AspectJ, but with only pure pointcuts. The calculus is untyped. The reader may have noticed that we do not model the join point stack in this paper. This is because it is not *required* for a given model of AOP to work. In fact, the join point stack is useful only to express control flow pointcuts. In our approach, this is achieved by specifying a user-defined pointcut designator for control flow, which uses a monad to thread the join point stack (or, depending on the desired level of dynamicity, a simple control flow state [19]). Support for the join point stack does not have to be included as a primitive in the core language. This is in fact how AspectJ is implemented.

Hofer and Osterman [12] shed some light on the modularity benefits of monads and aspects, clarifying that they are different mechanisms with quite different features: monads do not support declarative quantification, and aspects do not provide any support for encapsulating computational effects. In this regard, our work does not attempt at unifying monads and aspects, contrary to what De Meuter suggested. Instead, we exploit monads in a given language to build a flexible embedding of aspects that can be modularly extended. In addition, the fully-typed setting provides the basis for reasoning about monadic effects.

The notion of *monadic weaving* was described by Tabareau [30], where he shows that writing the aspect weaver in a monadic style paves the way for modular language extensions. He illustrated the extensibility approach with execution levels [32] and level-aware exception handling [10]. The authors then worked on a practical monadic aspect weaver in Typed Racket [11]. However, the type system of Typed Racket turned out to be insufficiently expressive, and the top type `Any` had to be used to describe pointcuts and advices. This was the original motivation to study monadic weaving in Haskell. Also in contrast to this work, prior work on monadic aspect weaving does not consider a base language with monads. In this paper, both the base language and the aspect weaver are monadic, combining the benefits of type-based reasoning about effects (Section 5) and modular language extensions (Section 6)—including type-based reasoning about language extensions.

Haskell has already been the subject of AOP investigations using the type class system as a way to perform static weaving [29]. AOP idioms are translated to type class instances, and type class resolution is used to perform static weaving. This work only supports simple pointcuts, pure aspects and static weaving, and is furthermore very opaque to modular changes as the translation of AOP idioms is done internally at compile time.

The specific flavor of pointcut/advice AOP that we developed is directly inspired by AspectScheme [9] and AspectScript [33]: dynamic aspect deployment, first-class aspects, and extensible set of pointcut designators. While we have not yet developed the more advanced scoping mechanisms found in these languages [31], we believe there is no specific challenges in this regard. The key difference here is that these languages are both dynamically typed, while we have managed to reconcile this high level of flexibility with static typing.

In terms of statically-typed functional aspect languages, the closest proposal to ours is AspectML [6]. In AspectML, pointcuts are first-class, but advice is not. The set of pointcut designators is fixed, as in AspectJ. AspectML does not support: advising anonymous functions, aspects of aspects, separate aspect deployment, and undeployment. AspectML was the first language in which first-class pointcuts were statically typed. The typing rules rely on anti-unification, just like we do in this paper. The major difference, though, is that AspectML is defined as a completely new language, with a specific type system and a specific core calculus. Proving type soundness is therefore very involved [6]. In contrast, we do not need to define a new type system and a new core calculus. Type soundness in our approach is derived straightforwardly from the type class that establishes the anti-unification relation. Half of section 4 is dedicated to proving that this type class is correct. Once this is done (and it is a result that is independent from AOP), proving aspect safety is direct. Another way to see this work is as a new illustration of the expressive power of the type system of Haskell, in particular how phantom types and type classes can be used in concert to statically type embedded languages.

Aspectual Caml [20] is another polymorphic aspect language. Interestingly, Aspectual Caml uses type information to influence matching, rather than for reporting type errors. More precisely, the type of pointcuts is inferred from the associated advices, and pointcuts only match join points that are valid according to these inferred types. We believe this approach can be difficult for programmers to understand, because it combines the complexities of quantification with those of type inference. Aspectual Caml is implemented by modifying the Objective Caml compiler, including modifications to the type inference mechanism. There is no proof of type soundness.

The advantages of our typed embedding do not only lie within the simplicity of the soundness proof. They can also be observed at the level of the implementation. The AspectML implementation is over 15,000 lines of ML code [6], and the Aspectual Caml implementation is around 24,000 lines of Objective Caml code [20]. In contrast, our implementation, including the execution levels extension (Section 6), is only 1,600 lines of Haskell code. Also, embedding an AOP extension entirely inside a mainstream language has a number of practical advantages, especially when it comes to efficiency and maintainability of the extension.

Finally, reasoning about advice effects has been studied from different angles. For instance, harmless advice can change termination behavior and use I/O, but no more [5]. A type and effect system is used to ensure conformance. Translucid contracts use grey box specifications and structural refinement in verification to reason about control effects [4]. In this work, we rather follow the type-based approach of EffectiveAdvice (EA) [23], which also accounts for various control effects and arbitrary computational effects. A limitation of EA is its lack of support quantification, relying instead on open recursive functions. A contribution of this work is to show how to extend this approach to the pointcut/advice mechanism. The subtlety lies in properly typing pointcuts. An interesting difference between both approaches is that in EA, it is not possible to talk about "the effects of all applied advices". Once an advice is composed with a base function, the result is seen as a base function for the following advice. In contrast, our approach, thanks to the aspect environment and dynamic weaving, makes it possible to keep aspects separate and ensure base/aspect separation at the effect level even in presence of multiple aspects. We believe that this splitting of the monadic stack is more consistent with programmers expectations.

9. Conclusion

We develop a novel approach to embed aspects in an existing language. We exploit monads and the Haskell type system to define a typed monadic embedding that supports both modular language extensions and reasoning about effects with pointcut/advice aspects. We show and prove how to ensure type soundness by design, including in presence of user-extensible pointcut designators, relying on a novel type class for establishing anti-unification. Compared to other approaches to statically-typed polymorphic aspect languages, the proposed embedding is more lightweight, expressive, extensible, and amenable to interference analysis. The approach can com-

bine Open Modules and EffectiveAdvice, and supports type-based reasoning about modular language extensions.

Acknowledgments

This work was supported by the INRIA Associated team RAPIDS.

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In A. P. Black, editor, *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, number 3586 in Lecture Notes in Computer Science, pages 144–168, Glasgow, UK, July 2005. Springer-Verlag.

[2] *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*, Brussels, Belgium, Apr. 2008. ACM Press.

[3] *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, Rennes and Saint Malo, France, Mar. 2010. ACM Press.

[4] M. Bagherzadeh, H. Rajan, G. T. Leavens, and S. Mooney. 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 2011)*, Porto de Galinhas, Brazil, Mar. 2011. ACM Press.

[5] D. S. Dantas and D. Walker. Harmless advice. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*, pages 383–396, Charleston, South Carolina, USA, Jan. 2006. ACM Press.

[6] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems*, 30(3):Article No. 14, May 2008.

[7] B. De Fraine, M. Südholt, and V. Jonckers. StrongAspectJ: flexible and safe pointcut/advice bindings. In AOSD 2008 [2], pages 60–71.

[8] R. Douence, P. Fradet, and M. Südholt. Trace-based aspects. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 201–217. Addison-Wesley, Boston, 2005.

[9] C. Dutchyn, D. B. Tucker, and S. Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, Dec. 2006.

[10] I. Figueroa and É. Tanter. A semantics for execution levels with exceptions. In *Proceedings of the 10th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2011)*, pages 7–11, Porto de Galinhas, Brazil, Mar. 2011. ACM Press.

[11] I. Figueroa, É. Tanter, and N. Tabareau. A practical monadic aspect weaver. In *Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2012)*, pages 21–26, Potsdam, Germany, Mar. 2012. ACM Press.

[12] C. Hofer and K. Ostermann. On the relation of aspects and monads. In *Proceedings of AOSD Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007)*, pages 27–33, 2007.

[13] M. P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP '00, pages 230–244, London, UK, UK, 2000. Springer-Verlag.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.

[16] D. Leijen and E. Meijer. Domain specific embedded compilers. In T. Ball, editor, *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages*, pages 109–122, 1999.

[17] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 333–343, New York, NY, USA, 1995. ACM.

[18] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS'03)*, volume 2895 of *Lecture Notes in Computer Science*, pages 105–121, Nov. 2003.

[19] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.

[20] H. Masuhara, H. Tatsuzawa, and A. Yonezawa. Aspectual Caml: an aspect-oriented functional language. In *Proceedings of the 10th ACM SIGPLAN Conference on Functional Programming (ICFP 2005)*, pages 320–330, Tallin, Estonia, Sept. 2005. ACM Press.

[21] W. D. Meuter. Monads as a theoretical foundation for aop. In *In International Workshop on Aspect-Oriented Programming at ECOOP*, page 25. Springer-Verlag, 1997.

[22] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991.

[23] B. C. d. S. Oliveira, T. Schrijvers, and W. R. Cook. EffectiveAdvice: discplined advice with explicit effects. In AOSD 2010 [3], pages 109–120.

[24] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[25] G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

[26] J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5:135–151, 1970.

[27] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the 12th ACM Symposium on Foundations of Software Engineering (FSE 12)*, pages 147–158. ACM Press, 2004.

[28] T. Sheard and S. P. Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, Dec. 2002.

[29] M. Sulzmann and M. Wang. Aspect-oriented programming with type classes. In *Proceedings of the 6th workshop on Foundations of aspect-oriented languages*, FOAL '07, pages 65–74, New York, NY, USA, 2007. ACM.

[30] N. Tabareau. A monadic interpretation of execution levels and exceptions for aop. In É. Tanter and K. J. Sullivan, editors, *Proceedings of the 11th International Conference on Aspect-Oriented Software Development (AOSD 2012)*, Potsdam, Germany, Mar. 2012. ACM Press.

[31] É. Tanter. Expressive scoping of dynamically-deployed aspects. In AOSD 2008 [2], pages 168–179.

[32] É. Tanter. Execution levels for aspect-oriented programming. In AOSD 2010 [3], pages 37–48.

[33] R. Toledo, P. Leger, and É. Tanter. AspectScript: Expressive aspects for the Web. In AOSD 2010 [3], pages 13–24.

[34] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages (POPL 89)*, pages 60–76, Austin, TX, USA, Jan. 1989. ACM Press.

[35] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, Sept. 2004.