Aspect-Oriented Programming with Type Classes

Martin Sulzmann
School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
sulzmann@comp.nus.edu.sg

Meng Wang
School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
wangmeng@comp.nus.edu.sg

Abstract

We study aspect-oriented programming (AOP) in the context of the strongly typed language Haskell. We show how to support AOP via a straightforward type class encoding. Our main result is that type-directed static weaving of AOP programs can be directly expressed in terms of type class resolution – the process of typing and translating type class programs. We provide two implementation schemes. One scheme is based on type classes as available in the Glasgow Haskell Compiler. The other, more expressive scheme, relies on an experimental type class system. Our results shed new light on AOP in the context of languages with rich type systems.

1. Introduction

Aspect-oriented programming (AOP) is an emerging paradigm which supports the interception of events at run-time. The essential functionality provided by an aspect-oriented programming language is the ability to specify *what* computation to perform as well as *when* to perform the computation. A typical example is profiling where we may want to record the size of the function arguments (what) each time a certain function is called (when). In AOP terminology, what computation to perform is referred to as the *advice* and when to perform the advice is referred to as the *pointcut*. An *aspect* is a collection of advice and pointcuts belonging to a certain task such as profiling.

There are numerous works which study the semantics of aspect-oriented programming languages, for example consider [2, 13, 26, 27, 29]. Some researchers have been looking into the connection between AOP and other paradigms such as generic programming [30]. To the best of our knowledge, we are the first to study the connection between AOP and the concept of type classes, a type extension to support ad-hoc polymorphism [25, 12], which is one of the most prominent features of Haskell [15].

In this paper, we make the following contributions:

- We introduce an AOP extension of Haskell, referred to as AOP Haskell, with type-directed pointcuts. Novel features of AOP Haskell include the ability to advise overloaded functions and refer to overloaded functions in advice bodies.
- We define AOP Haskell by means of a syntax-directed translation scheme where AOP programming idioms are directly expressed in terms of type classes. Thus, typing and translation of AOP Haskell can be explained in terms of typing and translation of the resulting type class program.
- We consider two possible implementation schemes. One scheme is based on type classes as supported by the Glasgow Haskell Compiler (GHC) [5]. We critically rely on multi-parameter type classes and overlapping instances. This scheme has restrictions in case we advise type annotated functions (Section 4).
- We show that these restrictions can be lifted by using a more flexible form of type classes as proposed by Stuckey and the first author [19]. We provide the type-directed translation rules from the more flexible AOP Haskell system to a simple target language. We establish concise results such as type soundness, type inference and coherence of the translation. These results can be directly related to existing results for type classes (Section 5).

We continue in Section 2 where we give an introduction to type classes. Section 3 gives an overview of the key ideas behind our approach of mapping AOP to type classes. We conclude in Section 6 where we also discuss related work.

2. Background: Type Classes

Type classes [12, 25] provide for a powerful abstraction mechanism to deal with user-definable overloading also known as ad-hoc polymorphism. The basic idea behind type classes is simple. Class declarations allow one to group together related methods (overloaded functions). Instance declarations prove that a type is in the class, by providing appropriate definitions for the methods.

Here are some standard Haskell declarations.

```
class Eq a where (==)::a->a->Bool
instance Eq Int where (==) = primIntEq      -- (I1)
instance Eq a => Eq [a] where               -- (I2)
   (==) [] [] = True
   (==) (x:xs) (y:ys) = (x==y) && (xs==ys)  -- (L)
   (==) _ _ = False
```

The class declaration in the first line states that every type `a` in *type class* `Eq` has an equality function `==`. Instance (I1) shows that `Int` is in `Eq`. We assume that `primIntEq` is the (primitive) equality function among `Int`s. The common terminology is to express membership of a type in a type class via constraints. Hence, we say that the *type class constraint* `Eq Int` holds. Instance (I2) shows that `Eq [a]` from the instance *head* holds if `Eq a` in the instance *context* holds. Thus, we can describe an infinite family of (overloaded) equality functions.

We can extend the type class hierarchy by introducing new subclasses.

```
class Eq a => Ord a where (<)::a->a->Bool   -- (S1)
instance Ord Int where ...                  -- (I3)
instance Ord a => Ord [a] where ...         -- (I4)
```

The above class declaration introduces a new subclass `Ord` which inherits all methods of its superclass `Eq`. For brevity, we ignore the straightforward instance bodies.

In the standard type class translation approach we represent each type class via a dictionary [25, 6]. These dictionaries hold the actual method definitions. Each superclass is part of its (direct) subclass dictionary. Instance declarations imply dictionary constructing functions and (super) class declarations imply dictionary extracting functions. Here is the dictionary translation of the above declarations.

```
type DictEq a = (a->a->Bool)
instI1 :: DictEq Int
instI1 = primIntEq
instI2 :: DictEq a -> DictEq [a]
instI2 dEqa =
 let eq [] [] = True
     eq (x:xs) (y:ys) = (dEqa x y) &&
                        (instI2 dEqa xs ys)
     eq _ _ = False
 in eq
type DictOrd a = (DictEq a, a->a->Bool)
superS1 :: DictOrd a -> DictEq a
superS1 = fst
instI3 :: DictOrd Int
instI3 = ...
instI4 :: DictOrd a -> DictOrd [a]
instI4 = ...
```

Notice how the occurrences of == on line (L) have been replaced by some appropriate dictionary values. For example, in the source program the expression `xs == ys` gives rise to the type class constraint `Eq [a]`. In the target program, the dictionary `instI2 dEqa` provides evidence for `Eq [a]` where `dEqa` is the (turned into a function argument) dictionary for `Eq a` and `instI2` is the dictionary construction function belonging to instance (I2).

The actual translation of programs is tightly tied to type inference. When performing type inference, we reduce type class constraints with respect to the set of superclass and instance declarations. This process is known as *type class resolution* (also known as context reduction). For example, assume some program text gives rise to the constraint `Eq [[a]]`. We reduce `Eq [[a]]` to `Eq a` via (reverse) application of instance (I2). Effectively, this tells us that given a dictionary `d` for `Eq a`, we can build the dictionary for `Eq [[a]]` by applying `instI2` twice. That is, `instI2 (instI2 d)` is the demanded dictionary for `Eq [[a]]`. Notice that given the dictionary `d'` for `Ord a`, we can build the alternative dictionary `instI2 (instI2 (superS1 d'))` for `Eq [[a]]`.

In the above, we only use single-parameter type classes. Other additional type class features include functional dependency [10], constructor [9] and multi–parameter [11] type classes. For the translation of AOP Haskell to Haskell we will use multi-parameter type classes and overlapping instances, yet another type class feature, as supported by GHC [5]. As we will see, GHC-style type classes have some limitations. Instead, we will later use a more flexible form of type classes which are an instance of our own general type class framework [19].

3. The Key Ideas

3.1 AOP Haskell

AOP Haskell extends the Haskell syntax [15] by supporting top-level aspect definitions of the form

```
N@advice #f1,...,fn# :: C => t = e
```

where `N` is a distinct label attached to each advice and the pointcut `f1`,...,`fn` refers to a set of (possibly overloaded) functions. Commonly, we refer to `fi`'s as *joinpoints*. Notice that our pointcuts are type-directed. Each pointcut has a type annotation `C => t` which follows the Haskell syntax. We refer to `C => t` as the *pointcut type*. We will apply the advice if the type of a joinpoint `fi` is an instance of `t` such that constraints `C` are satisfied. The advice body `e` follows the Haskell syntax for expressions with the addition of a new keyword `proceed` to indicate continuation of the normal evaluation process. We only support "around" advice which is sufficient to represent "before" and "after" advice.

In Figure 1, we give an example program. In the top part, we provide the implementation of an insertion sort algorithm where elements are sorted in non-decreasing order. At some stage during the implementation, we decide to add some security and optimization aspects to our implementation. We want to ensure that each call to `insert` takes a sorted list as an input argument and returns a sorted list as the result.

```
import List(sort)

insert x [] = [x]
insert x (y:ys)
  | x <= y    = x:y:ys
  | otherwise = y : insert x ys

insertionSort [] = []
insertionSort xs =
  insert (head xs) (insertionSort (tail xs))

-- sortedness aspect
N1@advice #insert# :: Ord a => a -> [a] -> [a] =
  \x -> \ys ->
     let zs = proceed x ys
     in if (isSorted ys) && (isSorted zs)
        then zs else error "Bug"
  where
     isSorted xs = (sort xs) == xs
-- efficiency aspect
N2@advice #insert# :: Int -> [Int] -> [Int] =
  \x -> \ys ->
     if x == 0 then x:ys
     else proceed x ys
```

Figure 1. AOP Haskell Example

In our AOP Haskell extension, we can guarantee this property via the first aspect definition in Figure 1. We make use of the (trusted) library function `sort` which sorts a list of values. The `sort` functions assumes the overloaded comparison operator `<=` which is part of the `Ord` class. Hence, we find the pointcut type `Ord a=>[a]->[a]->[a]`. The keyword `proceed` indicates to continue with the normal evaluation. That is, we continue with the call `insert x ys`. The second aspect definition provides for a more efficient implementation in case we call `insert` on list of `Int`s. We assume that only non-negative numbers are sorted which implies that `0` is the smallest element appearing in a list of `Int`s. Hence, if `0` is the first element it suffices to cons `0` to the input list. Notice there is an overlap among the pointcut types for `insert`. In case we call `insert` on list of `Int`s we apply both advice bodies in no specific order unless otherwise stated. For all other cases, we only apply the first advice.

A novel feature of AOP Haskell is that advice bodies may refer to overloaded functions. See the first advice body where we make use of the (overloaded) equality operator == whose type is `Eq a => a -> a -> a`. In Haskell, the `Eq` class is a superclass of `Ord`. Hence, there is no need to mention the `Eq` class in the pointcut type of the advice definition. Besides ordinary function, we can advise

- overloaded functions,
- polymorphic recursive functions, and
- functions appearing in advice and instance bodies.

We will see such examples later in Section 3.4 and 4.2.

3.2 Typing and Translating AOP Haskell with Type Classes

Our goal is to embed AOP Haskell into Haskell by making use of Haskell's rich type system. We seek a transformation scheme where typing and translation of the *source* AOP Haskell program is described by the resulting *target* Haskell program.

```
insert x [] = [x]
insert x (y:ys)
 | x <= y   = x:y:ys
 | otherwise=
 y : (joinpoint N1 (joinpoint N2 insert)) x ys --(1)

insertionSort [] = []
insertionSort xs =
 (joinpoint N1 (joinpoint N2 insert)) --(2)
 (head xs) (insertionSort (tail xs))

-- translation of advice
class Advice n t where
 joinpoint :: n -> t -> t
 joinpoint _ = id      -- default
data N1 = N1
instance Ord a => Advice N1 (a->[a]->[a]) where -- (I1)
joinpoint N1 insert =
 \x -> \ys -> let zs = insert x ys
             in if (isSorted ys) && (isSorted zs)
                then zs else error "Bug"
 where
    isSorted xs = (sort xs) == xs
instance Advice N1 a -- (I1') default case

data N2 = N2
instance Advice N2 (Int->[Int]->[Int]) where -- (I2)
 joinpoint N2 insert = \x -> \ys ->
             if x == 0 then x:ys
             else insert x ys
instance Advice N2 a -- (I2') default case
```

Figure 2. GHC Haskell Translation of Figure 1

The challenge we face is how to intercept calls to joinpoints and redirect the control flow to the advice bodies. In AOP terminology, this process is known as aspect weaving. Weaving can either be performed dynamically or statically. Dynamic weaving is the more flexible approach. For example, aspects can be added and removed at run-time. For AOP Haskell, we employ static weaving which is more restrictive but allows us to give stronger static guarantees about programs.

Our key insight is that type-directed static weaving can be phrased in terms of type classes based on the following principles:

- We employ type class instances to represent advice.
- We use a syntactic pre-processor to instrument joinpoints with calls to overloaded "weaving" function.
- We explain type-directed static weaving as type class resolution. Type class resolution refers to the process of reducing type class constraints with respect to the set of instance declarations.

In Figure 2, we transform the AOP Haskell program from Figure 1 to Haskell based on the first two principles. We use here type classes as supported by GHC.

Let us take a closer look at how this transformation scheme works. First, we introduce a two-parameter type class `Advice` which comes with a method `joinpoint`. Each call to `insert` is replaced by

```
joinpoint N1 (joinpoint N2 insert)
```

We assume here the following order among advice: $\mathtt{N2} \leq \mathtt{N1}$. That is, we first apply the advice `N1` before applying advice `N2`. This transformation step requires to traverse the abstract syntax tree and can be automated by pre-processing tools such as Template Haskell [18].

Each advice is turned into an instance declaration where the type parameter `n` of the `Advice` class is set to the singleton type of the advice and type parameter `t` is set to the pointcut type. In case the pointcut type is of the form `C => ...`, we set the instance context to `C`. See the translation of advice `N1`. In the instance body, we simply copy the advice body where we replace `proceed` by the name of the advised function. For each advice `N`, we add `instance Advice N a` where the body of this instance is set to the default case as specified in the class declaration. The reader will notice that for each advice we create two "overlapping" instances. That is, the instance heads overlap, hence, we can potentially use either of the two instances to resolve a type class constraint which may yield to two different results. We come back to this point shortly.

The actual (static) weaving of the program is performed by the type class resolution mechanism. GHC will infer the following types for the transformed program.

```
insert :: forall a.
          (Advice N1 (a -> [a] -> [a]),
           Advice N2 (a -> [a] -> [a]),
           Ord a) =>
           a -> [a] -> [a]

insertionSort :: forall a.
          (Advice N1 (a -> [a] -> [a]),
           Advice N2 (a -> [a] -> [a]),
           Ord a) =>
           a -> [a] -> [a]
```

Each `Advice` type class constraint results from a call to `joinpoint`. GHC does not resolve `Advice N1 (a -> [a] -> [a])` because we could either apply instance (I1) or the default instance (I1') which may yield to an ambiguous result. We say that GHC applies a "lazy" type class resolution strategy. However, if we use `insert` or `insertionSort` in a specific monomorphic context we can resolve "unambiguously" the above constraints.

Let us assume we apply `insertionSort` to a list of `Int`s. Then, we need to resolve the constraints

```
(Advice N1 (Int -> [Int] -> [Int]),
 Advice N2 (Int -> [Int] -> [Int]), Ord Int)
```

GHC applies the "best-fit" strategy and resolves `Advice N1 (Int -> [Int] -> [Int])` via instance (I1), `Advice N2 (Int -> [Int] -> [Int])` via instance (I2) and `Ord Int` is resolved using a pre-defined instance from the Haskell Prelude [15]. Effectively, this means that at locations (1) and (2) in the above program text, we intercept the calls to `insert` by first applying the body of instance (I1) followed by applying the body of instance (I2)

In case, we apply `insertionSort` to a list of `Bool`s, we need to resolve the constraints

```
(Advice N1 (Bool -> [Bool] -> [Bool]),
 Advice N2 (Bool -> [Bool] -> [Bool]), Ord Bool)
```

The instance (I1) is still the best-fit for `Advice N1 (Bool -> [Bool] -> [Bool])`. However, instead of instance (I2) we apply the default case to resolve `Advice N2 (Bool -> [Bool] -> [Bool])`. Hence, at locations (1) and (2) we apply the body of instance (I1) followed by the body of the default instance for advice (I2). `Ord Bool` is resolved using a pre-defined instance from the Haskell Prelude.

Figure 3 summarizes our approach of typing and translating AOP Haskell. In Section 4.1, we formalize AOP Haskell as a domain-specific extension of Haskell using GHC style type classes. Unfortunately, the system as described so far has some short-comings

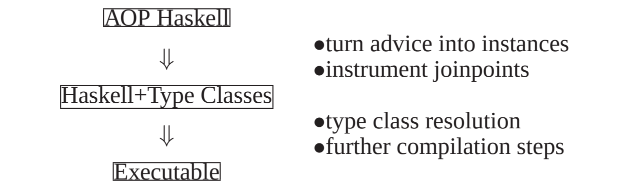

Figure 3. AOP Haskell Typing and Translation Scheme

```
f :: [a] -> Bool
f [] = True
f (x:xs) = f [xs]

N@advice ♯f♯ :: [[Bool]] -> Bool = \x -> False
```

Figure 4. Advising Polymorphic Recursive Functions

in case joinpoints are enclosed by type annotations. The short-comings are due to the way type classes are implemented in GHC. We can solve the problem by using an alternative type class system. Next, we will first examine the problem with GHC type classes and then we consider the alternative type class system.

3.3 Short-comings using GHC Style Type Classes

Let us assume we provide explicit type annotations to the functions in Figure 1.

```
insert :: Ord a => a -> [a] -> [a]
insertionSort :: Ord a => [a] -> [a]
```

The trouble is that if we keep `insert`'s annotation in the resulting target program, we find some unexpected (un-aspect like) behavior. GHC's type class resolution mechanism will "eagerly" resolve the constraints

```
Advice N1 (a -> [a] -> [a]),
Advice N2 (a -> [a] -> [a])
```

arising from

```
joinpoint N1 (joinpoint N2 insert)
```

by applying instance (I1) on `Advice N1 (a -> [a] -> [a])` and applying the default instance (I2') on `Advice N2 (a -> [a] -> [a])`. Hence, will never apply the advise `N2`, even if we call insert on list of `Int`s.

The conclusion is that we must either remove type annotations in the target program, or appropriately rewrite them during the translation process. For example, in the translation we must rewrite `insert`'s annotation to

```
insert :: (Advice N1 (a -> [a] -> [a]),
           Advice N2 (a -> [a] -> [a]), Ord a) =>
           a -> [a] -> [a]
```

At first look, this does not seem to be a serious limitation (rather tedious in case we choose to rewrite type annotations). However, there are type extensions which critically rely on type annotations. For example, consider polymorphic recursive functions which demand type annotations to guarantee decidable type inference [8]. In such cases we are unable to appropriately rewrite type annotations if we rely on GHC type classes to encode AOP Haskell.

Let us consider a (contrived) program to explain this point in more detail. In Figure 4, function `f` makes use of polymorphic recursion in the second clause. We call `f` on list of lists whereas the argument is only a list. Function `f` will not terminate on any argument other than the empty list. The advice definition allows us to intercept all

```
f :: [a] -> Bool
f [] = True
f (x:xs) = (joinpoint N f) [xs]

class Advice n t where
  joinpoint :: n -> t -> t
  joinpoint _ = id -- default case

data N = N
instance Advice N ([[Bool]]->Bool) where
  joinpoint N = \x -> False
instance Advice N a
```

Figure 5. GHC Haskell Translation of Figure 4

calls to `f` on list of list of `Bool`s to ensure termination for at least some values.

To translate the above AOP Haskell program to Haskell with GHC type classes we cannot omit `f`'s type annotation because `f` is a polymorphic recursive function. Our only hope is to rewrite `f`'s type annotation. For example, consider the attempt.

```
f :: Advice N a => [a] -> Bool
f [] = True
f (x:xs) = (joinpoint N f) [xs]
```

The call to `f` in the function body gives rise to `Advice N [a]` whereas the annotation only supplies `Advice N a`. Therefore, the GHC type checker will fail. Any similar "rewrite" attempt will lead to the same result (failure).

A closer analysis shows that the problem we face is due to the way type classes are implemented in GHC. In GHC, type classes are translated using the dictionary-passing scheme [6] where each type class is represented by a dictionary containing the method definitions. In our case, dictionaries represent the advice which will be applied to a joinpoint. Let us assume we initially call `f` with a list of `Bool`s. Then, the default advice applies and we proceed with `f`'s evaluation. Subsequently, we will call `f` on a list of list of `Bool`s. Recall that `f` is a polymorphic recursive function. Now, we wish that the advice `N` applies to terminate the evaluation with result `False`. The problem becomes now clear. The initial advice (i.e. dictionary) supplied will need to be changed during the evaluation of function `f` We cannot naturally program this behavior via GHC type classes.

3.4 More Flexible Type Classes for AOP Haskell

The solution we propose is to switch to an alternative type class translation scheme to translate advice. Instead of dictionaries we pass around types and select the appropriate method definitions (i.e. advice) based on run-time type information. Then, we can apply the straightforward AOP to type class transformation. In an intermediate step, the program from Figure 4 translates to the program in Figure 5. Whereas GHC fails to type check and compile the program in Figure 5, we can type check and compile this program under a type-passing based type class resolution scheme. The formal details are described in Section 5.4. The resulting program is given in Figure 6. We use a target language extended with a form of type case similar to intensional type analysis [7]. Based on the run-time type information, we call the appropriate advice.

3.5 Outline of The Rest of The Paper

In the upcoming section, we show how to express a "light-weight" form of AOP using GHC multi-parameter type classes and overlapping instances. Programming in AOP Haskell light has some restrictions. In case joinpoints are enclosed by type annotations,

```
f = Λ a. \ys:[a] ->
      case ys of
        [] -> True
        (x:xs) -> (joinpoint N ([[a]]->Bool)
                  (f [a])) [xs]
joinpoint = Λ n. Λ a. typecase (n,a) of
              (N,[[b]]->Bool) -> \f -> \x -> False
              (N,_) -> \f -> f
```

Figure 6. Type-Passing Type Class Resolution Applied to Figure 5

we must remove these annotations which is not possible in case of polymorphic recursive functions.

In Section 5, we show how to lift these restrictions by employing a type-passing type class translation scheme. The type class system necessary to describe "full" AOP Haskell is an instance of the general type class framework proposed by Stuckey and the first author [19]. In particular, we can derive strong results for AOP Haskell such as type inference and coherence via reduction from known results for type classes.

4. AOP Haskell Light in GHC

We consider an extension of GHC with top-level aspect definitions of the form

```
N@advice #f1,...,fn# :: C => t = e
```

We omit to give the syntactic description of Haskell programs which can be found elsewhere [15]. We assume that type annotation `C => t` and expression `e` follow the Haskell syntax (with the addition of a new keyword `proceed` which may appear in `e`). We assume that symbols `f1`,...,`fn` refer to the names of (top-level) functions and methods (i.e. overloaded functions). See also Section 3.1.

As motivated in Section 3.3, we impose the following condition on the AOP extension of GHC.

DEFINITION 1 (AOP Haskell Light Restriction). *We demand that each joinpoint* `f` *is not enclosed by a type annotation, advice or instance declaration.*

Notice that instance declarations "act" like type annotations. In the upcoming translation scheme we will translate advice declarations to instance declarations. Hence, joinpoints cannot be enclosed by advice and instance declarations either.

Next, we formalize the AOP to type class transformation scheme. We will conclude this section by providing a number of programs written in AOP Haskell light.

4.1 Type Class-Based Transformation Scheme

Based on the discussion in Section 3.2, our transformation scheme proceeds as follows.

DEFINITION 2 (AOP to Type Class Transformation Scheme). *Let p be an AOP Haskell program. We perform the following transformation steps on p to obtain the program p'.*

Advice class: *We add the class declaration*

```
class Advice n t where
      joinpoint :: n -> t -> t
      joinpoint _ = id
```

Advice bodies: *Each AOP Haskell statement*

```
N@advice #f1,...,fn# :: C => t = e
```

is replaced by

```
data N = N
instance C => Advice N t where
  joinpoint _ f = e'
instance Advice N a -- default case
```

where `e'` *results from* `e` *by substituting* `proceed` *by the fresh name* `f`.

Joinpoints: *For each function* `f` *and for all advice* `N1`*, ...,* `Nm` *where* `f` *appears in their pointcut we replace* `f` *by*

```
joinpoint N1 (... (joinpoint Nm f)...)
```

being careful to avoid name conflicts in case of lambda-bound function names. We assume that the order among advice is as follows: `Nm` $\leq \ldots \leq$ `N1`.

To compile the resulting program we rely on the following GHC extensions (compiler flags):

- `-fglasgow-exts`
- `-fallow-overlapping-instances`

The first flag is necessary because we use multi-parameter type classes. The second flag enables support for overlapping instances.

FACT 1. *Type soundness and type inference for AOP Haskell light are established via translation to GHC-style type classes.*

We take it for granted that GHC is type sound and type inference is correct. However, it is difficult to state any precise results given the complexity of Haskell and the GHC implementation. In Section 5, we will formally develop type soundness and type inference for a core fragment of AOP Haskell.

An assumption which we have not mentioned so far is that we can only advise function names which are in scope. That is, pointcuts and joinpoints must be in the same scope. We will explain this point by example in the next (sub)section.

Another issue is that in our current type class encoding of AOP we do not check whether advice definitions have any effect on programs. For example, consider

```
f :: Int
f = 1

N@advice #f# :: Bool = True
```

where the advice definition `N` is clearly useless. We may want to reject such useless definitions by adding the following transformation step to Definition 2.

Useful Advice: Each AOP Haskell statement

```
N@advice #f1,...,fn# :: C => t = e
```

generates

```
f1' :: C => t
f1' = f1
...
fn' :: C -> t
fn' = fn
```

in `p'` where `f1'`, ..., `fn'` are fresh identifiers.

FACT 2. *We find that definitions* `f1'`*,...,*`fn'` *are well-typed iff the types of* `f1`*,...,*`fn` *are more specific than the pointcut type* `C=>t`.

In case of our above example, we generate

```
f' :: Bool
f' = f
```

which is ill-typed. Hence, we reject the useless advice `N`.

```
accF xs acc = accF (tail xs) (head xs : acc)
reverse :: [a] -> [a] -> [a]
reverse xs = accF xs []
append :: [a] -> [a] -> [a]
append xs ys = accF xs ys

N@advice ♯accF♯ :: [a] -> [a] -> [a] =
   \xs -> \acc -> case xs of
                    [] -> acc
                    _ -> proceed xs acc
```

Figure 7. Advising Accumulator Recursive Functions

```
module CollectsLib where

class Collects c e | c -> e where
  insert :: e -> c -> c
  test :: e -> c -> Bool
  empty :: c

instance Ord a => Collects [a] a where
  insert x [] = [x]
  insert x (y:ys)
    | x <= y = x:y:ys
    | otherwise = y : (insert x ys)
  test x xs = elem x xs
  empty = []
```

Figure 8. Collection Library

4.2 AOP Haskell Light Examples

We take a look at a few AOP Haskell light example programs. We will omit the translation to (GHC) Haskell which can be found here [20]. We also discuss issues regarding the scope of pointcuts and how to deal with cases where the joinpoint is enclosed by an annotation.

Advising recursive functions. Our first example is given in Figure 7. We provide definitions of `append` and `reverse` in terms of the accumulator function `accF`. We deliberately left out the base case of function `accF`. In AOP Haskell light, we can catch the base case via the advice `N`. It is safe here to give append and reverse type annotations, although, the joinpoint is then enclosed by a type annotation. The reason is that only one advice `N` applies here.

Advising overloaded functions. In our next example, we will show that we can even advise overloaded functions. We recast the example from Section 3.1 in terms of a library for collections. See Figures 8 and 9. We use the functional dependency declaration `Collects c e | c->e` to enforce that the collection type `c` uniquely determines the element type `e`. We use the same aspect definitions from earlier on to advise function `insertionSort` and the now overloaded function `insert`. As said, we only advise function names which are in the same scope as the pointcut. Hence, our transformation scheme in Definition 2 effectively translates the code in Figure 9 to the code shown in Figure 2. The code in Figure 8 remains unchanged.

Advising functions in instance declarations. If we wish to advise all calls to `insert` throughout the entire program, we will need to place the entire code into one single module. Let us assume we replace the statement `import CollectsLib` in Figure 9 by the code in Figure 8 (dropping the statement `module CollectsLib where` of course). Then, we face the problem of advising a function enclosed by a "type annotation". Recall that instance declarations act like type annotations and there is now a joinpoint `insert` within the body of the instance declaration in scope. Our automatic transformation scheme in Definition 2 will not work here. The resulting program may type check but we risk that the program will show some unaspect-like behavior. The (programmer-guided) solution is to manually rewrite the instance declaration during the transformation process which roughly yields the following result

```
module Main where

import List(sort)
import CollectsLib

insertionSort [] = []
insertionSort xs =
  insert (head xs) (insertionSort (tail xs))

N1@advice ♯insert♯ :: Ord a => a -> [a] -> [a] =
  \x -> \ys ->
     let zs = proceed x ys
     in if (isSorted ys) && (isSorted zs)
        then zs else error "Bug"
  where
     isSorted xs = (sort xs) == xs
N2@advice ♯insert♯ :: Int -> [Int] -> [Int] =
  \x -> \ys -> if x == 0 then x:ys
               else proceed x ys
```

Figure 9. Advising Overloaded Functions

```
N1@advice ♯f♯ :: [Int] -> Int =
   \xs -> (head xs) + (proceed (tail xs))


N2@advice ♯head♯ :: [Int] -> Int =
   \xs -> case xs of
            [] -> -1
            _ -> proceed xs
```

Figure 10. Advising functions in advice bodies

```
...
instance (Advice N1 (a->[a]->[a]),
          Advice N2 (a->[a]->[a]),
          Ord a) => Collects [a] a where
 insert x [] = [x]
 insert x (y:ys)
  | x <= y    = x:y:ys
  | otherwise =
     y : ((joinpoint N2 (joinpoint N1 insert)) x ys)
...
```

To compile the transformed AOP Haskell light program with GHC, we will need to switch on the following additional compiler flag:

- `-fallow-undecidable-instances`

We would like to stress that type inference for the transformed program is decidable. The "decidable instance check" in GHC is simply conservative, hence, we need to force GHC to accept the program.

Advising functions in advice bodies. Given that we translate advice into instances, it should be clear that we can also advice functions in advice bodies if we are willing to "guide" the translation

```
type T = [Int] -> Int
data N1 = N1
instance Advice N2 T => Advice N1 T where
  joinpoint N1 f =
    \ xs -> ((joinpoint N2 head) xs) + (f (tail xs))

data N2 = N2
instance Advice N2 T where
  joinpoint N2 head =
    \xs -> case xs of
             [] -> -1
             _ -> head xs
```

Figure 11. GHC Haskell Translation of Figure 10

Types	t	$::=$	$a \mid t \rightarrow t \mid T\ \bar{t}$
Type Scheme	σ	$::=$	$t \mid \forall \bar{a}.C \Rightarrow t$
Type Classes	tc	$::=$	$TC\ t$
Constraints	C	$::=$	$tc_1 \wedge ... \wedge tc_n$
Expressions	e	$::=$	$\mathsf{proceed} \mid x \mid \lambda x.e \mid e\ e \mid$ $\mathsf{let}\ \begin{array}{l} x :: C \Rightarrow t \\ x = e \end{array}\ \mathsf{in}\ e$
Pointcut	pc	$::$	$x_1, ..., x_n$
Advice	adv	$::=$	$N@\mathsf{advice}\ \sharp pc\sharp :: C \Rightarrow t = e$
Classes	cl	$::=$	$\mathsf{class}\ TC\ \bar{a}\ \mathsf{where}\ m :: C \Rightarrow t$
Instances	$inst$	$::=$	$\mathsf{instance}\ C \Rightarrow\ TC\ \bar{t}\ \mathsf{where}\ m = e$
Programs	p	$::=$	$\overline{adv}; \overline{cl}; \overline{inst}; e$

Figure 12. AOP Mini Haskell Syntax

scheme. In Figure 10, we give such an example and its (manual) translation is given in Figure 10. We rely again on the "undecidable" instance extension in GHC.

The last example makes us clearly wish for a system where we do not have to perform any manual rewriting. Of course, we could automate the rewriting of annotations by integrating the translation scheme in Definition 2 with the GHC type inferencer. However, the problem remains that we are unable to advise polymorphic recursive functions. Recall the discussion in Section 3.3. Hence, we seek for a more principled AOP extension of Haskell.

Next, we formally define the semantics and type inference for an AOP extension of a core fragment of Haskell. We make use of more flexible type classes to translate AOP programming idioms. Thus, we obtain a more principled and powerful system where we can also advise polymorphic recursive functions and verify important formal results such as type inference and coherence of the translation.

5. AOP Mini Haskell

We first define the syntax of AOP Mini Haskell. We use the term "Mini" to indicate that we only consider a core fragment of Haskell. Then, we develop some technical machinery necessary to concisely describe the type-directed translation rules from AOP Mini Haskell to a simple target language. We use a type-passing scheme to translate type classes and advice. We conclude this section by stating some formal results.

5.1 Syntax

In Figure 12, we give the syntax of AOP Mini Haskell. We use the following conventions. We write $\bar{o}$ as a short-hand to denote a sequence of objects $o_1, ..., o_n$. We assume a distinct type class $True$ representing the always true constraint. We write $\forall \bar{a}.t$ as a short-hand for $\forall \bar{a}.True \Rightarrow t$. For simplicity, we ignore case expressions and assume that let-defined (possibly recursive) functions carry type annotations. We also assume that each class declaration introduces a type class with a single method only. In example programs we may make use of pattern matching syntax which can be expressed via primitives such as $\mathsf{head} : \forall \mathsf{a}.[\mathsf{a}] \rightarrow \mathsf{a}$ and $\mathsf{tail} : \forall \mathsf{a}.[\mathsf{a}] \rightarrow [\mathsf{a}]$ which are recorded in some initial environment Γ_{init}.

Before we define the semantics of AOP Mini Haskell, we first define the semantics of type classes. We also define a subsumption relation among types which is defined in terms of the type class semantics.

5.2 Type Class Semantics

We explain the meaning of type classes in terms of Stuckey's and the first author's type class framework [19]. The idea is to translate class and instance declarations into Constraint Handling Rules (CHRs) [4]. CHRs serve as a meta specification language to reason about type class relations.

For example, the instance declaration from Figure 8

```
instance Ord a => Collects [a] a
```

translates to the CHR

```
Collects [a] a <==> Ord a
```

Logically, the symbol `<==>` stands for bi-implication while the operational reading is to replace (i.e. rewrite) the constraints on the left-hand side by those on the right-hand side. In contrast to Prolog, we only perform matching but *not* unification during rule application.

The advantage of CHRs is that we can more concisely describe advice without having to resort to overlapping instances. Recall that in AOP Haskell light the advice declaration

```
N1@advice #insert# :: Ord a => a -> [a] -> [a] = ...
```

from Figure 1 translates to the overlapping instances

```
instance Ord a => Advice N1 (a->[a]->[a])
instance Advice N1 a
```

We then relied on GHC's "lazy" and "best-fit" type class resolution strategy to faithfully encode AOP.

In AOP Mini Haskell, we use CHRs with explicit guard constraints to express type class relations implied by advice declarations. For example, for the above example we generate the following CHRs.

```
Advice N1 (a->[a]->[a]) <==> Ord a
Advice N1 b <==> b /= (a->[a]->[a]) | True
```

The first CHR fires if we encounter a joinpoint of type `(t->[t]->[t])` which means that advice `N1` applies. The second CHR contains a guard constraint and therefore only applies to joinpoints which are *not* instances of `(t->[t]->[t])`. In this case advice `N1` does not apply. Hence, in the type class translation we use the default instance. The upshot of using CHRs with guard constraints is that they enable us to give a more concise (type class) description of advice including precise results (see upcoming Section 5.5).

We formalize the syntax and semantics of CHRs. We assume that *fv*(o) computes the free variables of some object o. For the moment, we are only concerned with the logical semantics of CHRs. We postpone the definition of the operational semantics until we discuss type inference.

DEFINITION 3 (CHR Syntax and Logical Semantics). *For our purposes, CHRs are of the form*

$$TC\ \bar{t}\ \iff\ \bar{t} \neq \overline{t'} \mid TC_1\ \overline{t_1}, ..., TC_n\ \overline{t_n}$$

Logically, we interpret the above as the first-order formula

$$\forall \bar{a}.((\exists \bar{b}.\bar{t} \neq \overline{t'}) \supset (TC\ \bar{t} \leftrightarrow \exists \bar{c}.(TC_1\ \overline{t_1} \wedge ... \wedge TC_n\ \overline{t_n})))$$

where $\bar{a} = fv(\bar{t})$, $\bar{b} = fv(\overline{t'}) - \bar{a}$ and $\bar{c} = fv(\overline{t_1}, ..., \overline{t_n}) - \bar{a}$. The above formula simplifies to $\forall \bar{a}.(TC\ \bar{t} \leftrightarrow \exists \bar{c}.(TC_1\ \overline{t_1} \wedge ... \wedge TC_n\ \overline{t_n}))$ in case we omit the guard constraint.

The full set of CHR is much richer and provides support for improvement conditions as implied by the functional dependency in Figure 8. We refer the interested reader to [3, 22] for details.

5.3 Subsumption

In the upcoming type-directed translation scheme, we employ a subsumption relation to compare types with respect to the *program logic* P. We assume that P contains the set of CHRs derived from type class declarations.

DEFINITION 4 (Subsumption). *Let P be a program logic and $\forall \bar{a}.C \Rightarrow t$ and $\forall \overline{a'}.C' \Rightarrow t'$ be two types. We define*

$$P \vdash (\forall \bar{a}.C \Rightarrow t) \leq (\forall \overline{a'}.C' \Rightarrow t')$$

iff $P \wedge C' \models \exists \bar{a}.(C \wedge t = t')$. We assume that there are no name clashes among $\bar{a}$ and $\overline{a'}$.

The statement $P \wedge C' \models \exists \bar{a}.(C \wedge t = t')$ holds if for any model M of $P \wedge C'$ (in the first-order sense) we find that $\exists \bar{a}.(C \wedge t = t')$ holds in M.

The important point to note is that the subsumption check turns into an entailment check among constraints. We postpone a discussion of how to operationally check for subsumption until we consider type inference.

5.4 Type-Directed Translation Scheme

We give the type-directed translation rules from AOP Mini Haskell to a simple target language. We slightly deviate from the scheme shown in Figure 3. Instead of first transforming AOP constructs to type class constructs and then translating type classes, we immediately translate AOP and type class constructs to the target language. It will be obvious how to split the upcoming *direct* translation scheme into a two-step translation scheme.

Figure 13 describes the target language. In the translation, we will write letrec $x = E_1$ in E_2 as syntactic sugar for let $x =$ (rec f in $[f/x]E_1$) E_2 where `f` is a fresh identifier. Multiple binding groups let $x_1 = E_1, ..., x_n = E_n$ in E can also be desugared into let $x = E'$ in E'' for some appropriate E' and E''. We also use type case, type application and type abstraction to support a type-passing type class resolution strategy. Again, these constructs are only syntactic sugar for value case, value application and value abstraction, assuming that types are represented by values. See the syntactic category TValue. We could easily switch to a "real" typed target language [7, 21] with no change in results.

We interpret target expressions in an untyped denotational semantics. The semantic equations are straightforward. For example, $+$ and $\sum$ denote coalesced sums and $\mathcal{V} \to \mathcal{V}$ is the continuous function space. In general, we leave injection and projection operators for sums implicit. The value $\mathbf{W}$ is the error element, $\mathcal{K}$ is the set of value constructors and `fix` refers to the fix-point operator. In the rule for case expressions, we write $\eta(v) = \eta_{v_i}(v_i)$ to denote that the value $\eta(v)$ is matched against v_i for some (local) value binding η_{v_i}. We write $\eta \cdot \eta_{v_i}$ to denote composition of value bindings. In the second semantic (type) equation from the bottom, we write $\Gamma^{target}_{init} \vdash K : \mu'_1 \to \ldots \to \mu'_n \to T\,\mu_1 \ldots \mu_m$ to denote a valid Hindley/Milner typing judgment.

Figure 14 specifies the translation of AOP Mini Haskell programs to target expressions in terms of five judgments of the form:

1. Programs: $p \vdash E$.
2. Preprocessing: $p \vdash \Gamma, P, J$.
3. Expressions: $C, \Gamma \vdash e : t \leadsto E$.

TType	tt	$::=$	$a \mid tt \to tt \mid T\ \overline{tt}$
TTypeScheme	σ_t	$::=$	$tt \mid \forall \bar{a}.tt$
TValue	v	$::=$	$a \mid v \to v \mid T\ \bar{v}$
Target	E	$::=$	$x \mid v \mid E\ E \mid \lambda x.E \mid \text{let}\ x = E\ \text{in}\ E \mid$ $\text{rec}\ f\ \text{in} E \mid \text{case}\ v\ \text{of}\ [\bar{v}_i \to E_i]_{i \in I}$

$$\mathcal{V} \ = \ \mathbf{W}_\perp + \mathcal{V} \to \mathcal{V} + \textstyle\sum_{K \in \mathcal{K}} (K\,\mathcal{V}_1 \ldots \mathcal{V}_{\text{arity}(K)})_\perp$$

$$\begin{array}{l} \eta : Var \to \mathcal{V} \\ [\![]\!] : Target \to (Var \to \mathcal{V}) \to \mathcal{V} \\ [\![]\!]_t : TTypeScheme \to \mathcal{V} \end{array}$$

$$\begin{array}{lcl}
[\![x]\!]\eta & = & \eta(x) \\
[\![\lambda x.E]\!]\eta & = & \lambda u.[\![E]\!]\eta[x := u] \\
[\![E\,E']\!]\eta & = & \text{if}\,[\![E]\!]\eta \in \mathcal{V} \to \mathcal{V} \\
& & \text{then}\ ([\![E]\!]\eta)\ ([\![E']\!]\eta) \\
& & \text{else}\ \mathbf{W} \\
[\![\text{let}\,x = E\,\text{in}\,E']\!]\eta & = & [\![E']\!]\eta[x := [\![E]\!]\eta] \\
[\![\begin{array}{l}\text{case}\,v\,\text{of} \\ [v_i \to E_i]_{i \in I}\end{array}]\!]\eta & = & \text{if}\,\eta(v) = \eta_{v_1}(v_1)\text{then}[\![E_1]\!](\eta \cdot \eta_{v_1}) \\
& & \ldots \\
& & \text{if}\,\eta(v) = \eta_{v_n}(v_n)\text{then}[\![E_n]\!](\eta \cdot \eta_{v_n}) \\
& & \text{else}\ \mathbf{W} \\
& & (\text{where}\ I = \{1, ..., n\}) \\
[\![\text{rec}\,f\,\text{in}\,E]\!]\eta & = & \mathtt{fix}(\lambda v.[\![E]\!]\eta[f := v])
\end{array}$$

$$\begin{array}{l}
[\![\mu_1 \to \mu_2]\!]_t = \\
\qquad \{\ f \in \mathcal{V} \to \mathcal{V} \mid x \in [\![\mu_1]\!]_t \Rightarrow f\,x \in [\![\mu_2]\!]_t\ \} \\
[\![T\,\mu_1 \ldots \mu_m]\!]_t = \\
\qquad \{\perp\} \cup \\
\qquad \bigcup\{K\,[\![\mu'_1]\!]_t \ldots [\![\mu'_n]\!]_t \mid \\
\qquad\qquad \Gamma^{target}_{init} \vdash K : \mu'_1 \to \ldots \to \mu'_n \to T\,\mu_1 \ldots \mu_m\} \\
[\![\forall \bar{a}.t]\!]_t = \\
\qquad \bigcap_{\bar{\mu}} [\![\overline{[\mu/a]}t]\!]_t
\end{array}$$

where for monotypes μ we require that $fv(\mu) = \emptyset$ and Γ^{target}_{init} contains the set of value constructors used in this context.

Figure 13. Target Language

4. Instances: $\Gamma \vdash \overline{adv} \leadsto jp = E$.
5. Advice: $\Gamma \vdash \overline{inst}_{TC} \leadsto m = E$.

From the previous (sub)section, we assume the subsumption judgment $P \vdash \sigma_1 \leq \sigma_2$ and the model-theoretic entailment relation $P \models C$.

The first judgment drives the translation process. In the premise of rule (Prog), we call the second judgment to collect the set Γ of method declarations implied by class declarations, the set P of CHRs implied by instance and advice declarations and the set J of pairs of function name and advice. As said, we omit the intermediate step where we first translate advice declarations into type class declarations. We directly translate advice declarations into CHRs using guard constraints to resolve the overlap among the advice N and the default case. We assume that $Advice$ is a special purpose type class (advice) constraint and for each advice N we find a value N of (singleton) type N in the initial environment Γ_{init}.

Then, we call the fourth and fifth judgment to translate the advise and instances. We write $\overline{inst}_{TC}$ to denote a sequence of instance declarations which refer to type class TC in their instance "head". The result is sequence of binding groups $jp = E', m_1 =$

$$\boxed{p \vdash E}$$

$$\text{(Prog)} \quad \frac{\begin{array}{c} \overline{adv}; \overline{cl}; \overline{inst}_{TC_1}, ..., \overline{inst}_{TC_n} \vdash \Gamma, P, J \\ \Gamma \cup \Gamma_{init} \vdash \overline{adv} \leadsto jp = E' \\ \Gamma \cup \Gamma_{init} \vdash \overline{inst}_{TC_i} \leadsto m_i = E_i \quad \text{for } i = 1, ..., n \\ C, \Gamma \cup \Gamma_{init} \vdash e : t \leadsto E \\ P \models C \end{array}}{\overline{adv}; \overline{cl}; \overline{inst}_{TC_1}, ..., \overline{inst}_{TC_n}; e \vdash \mathsf{let}\ jp = E', m_1 = E_1, ..., m_n = E_n\ \mathsf{in}\ E}$$

$$\boxed{p \vdash \Gamma, P, J}$$

$$\mathsf{instance}\ C \Rightarrow TC\ \bar{t}\ \mathsf{where}\ m = e \vdash \{TC\ \bar{t} \Longleftrightarrow C\} \qquad \frac{\overline{inst} \vdash P_1 \quad inst \vdash P_2}{\overline{inst}, inst \vdash P_1 \cup P_2}$$

$$\frac{P = \{Advice\ N\ t \Longleftrightarrow C, Advice\ N\ t \Longleftrightarrow a \neq t \mid True\}}{N@\mathsf{advice}\ \sharp f_1, ..., f_m \sharp :: C \Rightarrow t = e \vdash P, \{(f_1, N), ..., (f_m, N)\}} \qquad \frac{\overline{adv} \vdash P_1, J_1 \quad adv \vdash P_2, J_2}{\overline{adv}, adv \vdash P_1 \cup P_2, J_1 \cup J_2}$$

$$\frac{\bar{b} = fv(C, t) - \bar{a}}{\mathsf{class}\ TC\ \bar{a}\ \mathsf{where}\ m :: C \Rightarrow t \vdash \{m : \forall \bar{a}, \bar{b}. TC\ \bar{a} \wedge C \Rightarrow t\}} \qquad \frac{\overline{cl} \vdash \Gamma_1 \quad cl \vdash \Gamma_2}{\overline{cl}, cl \vdash \Gamma_1 \cup \Gamma_2}$$

$$\frac{\overline{adv} \vdash P_a, J \quad \overline{cl} \vdash \Gamma \quad \overline{inst} \vdash P_i}{\overline{adv}; \overline{cl}; \overline{inst}; e \vdash \Gamma, P_a \cup P_i, J}$$

$$\boxed{C, \Gamma \vdash e : t \leadsto E}$$

$$\text{(Abs)} \quad \frac{C, \Gamma \cup \{x : t_1\} \vdash e : t_2 \leadsto E}{C, \Gamma \vdash \lambda x.e : t_1 \to t_2 \leadsto \lambda x.E} \qquad \text{(App)} \quad \frac{C, \Gamma \vdash e_1 : t_2 \to t_1 \leadsto E_1 \quad C, \Gamma \vdash e_2 : t_2 \leadsto E_2}{C, \Gamma \vdash e_1\ e_2 : t_1 \leadsto E_1\ E_2}$$

$$\text{(Let)} \quad \frac{\begin{array}{c} C'_1, \Gamma \cup \{x : \forall \bar{a}. C_1 \Rightarrow t_1\} \vdash e_1 : t'_1 \leadsto E_1 \\ P \vdash (\forall \bar{b}. C'_1 \Rightarrow t'_1) \leq (\forall \bar{a}. C_1 \Rightarrow t_1) \quad \bar{a} = fv(C_1, t_1) \quad \bar{b} = fv(C'_1, t'_1) - fv(\Gamma) \\ C, \Gamma \cup \{x : \forall \bar{a}. C_1 \Rightarrow t_1\} \vdash e_2 : t_2 \leadsto E_2 \end{array}}{C, \Gamma \vdash \mathsf{let}\ \begin{array}{l} x :: C_1 \Rightarrow t_1 \\ x = e_1 \end{array}\ \mathsf{in}\ e_2 : t_2 \leadsto \mathsf{letrec}\ x = \Lambda \bar{a}. E_1\ \mathsf{in}\ E_2}$$

$$\text{(Var-JP)} \quad \frac{\begin{array}{c} (f :: \forall \bar{a}. C' \Rightarrow t') \in \Gamma \quad J(f) = \{(f, N_1), ..., (f, N_m)\} \\ t'' = [\overline{t/a}]t' \quad C = [\overline{t/a}]C' \wedge Advice\ N_1\ t'' \wedge ... \wedge Advice\ N_m\ t'' \end{array}}{C, \Gamma \vdash f : t'' \leadsto jp\ N_1\ t''\ (...(jp\ N_m\ t''(f\ \bar{t}))...)} \qquad \text{(Var-}\forall\text{Elim)} \quad \frac{(x : \forall \bar{a}. C' \Rightarrow t') \in \Gamma \quad J(x) = \emptyset}{C \wedge [\overline{t/a}]C', \Gamma \vdash x : [\overline{t/a}]t' \leadsto x\ \bar{t}}$$

$$\boxed{\Gamma \vdash \overline{adv} \leadsto jp = E}$$

$$\text{(Advice)} \quad \frac{\begin{array}{c} C'_i, \Gamma \cup \{f : t_i\} \vdash [f/\mathsf{proceed}]e_i : t'_i \leadsto E_i \quad f\ \text{fresh} \\ P \vdash (\forall \overline{b_i}. C'_i \Rightarrow t'_i) \leq (\forall \overline{c_i}. C_i \Rightarrow t_i) \quad \overline{b_i} = fv(C'_i, t'_i) - fv(\Gamma) \quad \overline{c_i} = fv(C_i, t_i) \quad \text{for } i \in I \end{array}}{\Gamma \vdash [N_i@\mathsf{advice}\ \sharp pc_i \sharp :: C_i \Rightarrow t_i = e_i]_{i \in I} \leadsto jp = \Lambda n. \Lambda a. \mathsf{typecase}\ (n, a)\ \mathsf{of}\ \begin{array}{l} (N_i, t_i) \to \lambda f. E_i \\ (N_i, \_) \to \lambda f. f \end{array}_{i \in I}}$$

$$\boxed{\Gamma \vdash \overline{inst}_{TC} \leadsto m = E}$$

$$\text{(Inst)} \quad \frac{\begin{array}{c} C'_i, \Gamma \vdash e_i : t'_i \leadsto E_i \quad (m : \forall \bar{a}, \bar{b}. TC\ \bar{a} \wedge C \Rightarrow t) \in \Gamma \quad \overline{d_i} = fv(\overline{t_i}) \\ P \vdash (\forall \overline{c_i}. C'_i \Rightarrow t'_i) \leq (\forall \overline{d_i}, \bar{b}. TC\ \overline{t_i} \wedge [\overline{t_i/a}]C \Rightarrow [\overline{t_i/a}]t) \quad \overline{c_i} = fv(C'_i, t'_i) - fv(\Gamma) \quad \text{for } i \in I \end{array}}{\Gamma, [\mathsf{instance}\ C_i \Rightarrow TC\ \overline{t_i}\ \mathsf{where}\ m = e_i]_{i \in I} \leadsto m = \Lambda \bar{a}. \mathsf{typecase}\ \bar{a}\ \mathsf{of}\ [\overline{t_i} \to \Lambda \bar{b}. E_i]_{i \in I}}$$

Figure 14. Mini AOP Haskell Translation Scheme

$E_1, ..., m_n = E_n$ defining the joinpoint and type class methods. Finally, we call the third judgment to translate the expression e. The condition $P \models C$ ensure that all type class and advise constraints arising from e are resolved.

The third judgment for translating expressions uses a constraint component C to infer the type class and advise constraints arising out of the program text. It should be clear that we could easily refine our formulation and infer also type equations (also known as unification constraints) in rules (Abs) and (App) on the expense of a more noisy presentation.

In rule (Var-∀Elim), we build an instance of x's type scheme for a given type. We write $[\overline{t/a}]$ to denote a substitution mapping variables a_i to types t_i. Rule (Var-JP) works similarly. In addition, we intercept the call to f and instrument the program text with calls to the advice defined for f. We write $J(f)$ as a short-hand for $\{(f', N) \in J \mid f = f'\}$. Recall that source types t are reflected in the target language as expressions v, but, we write t for simplicity.

In rule (Let), we deal with *closed* type annotated function definitions. That is, we quantify over the set of variables in $C \Rightarrow t$. In the translation of the body of the let functions, we may refer to the let function. Hence, we support (possibly polymorphic) recursive functions. We use the subsumption check to test whether the inferred type $\forall \bar{b}.C_1' \Rightarrow t_1'$ subsumes the annotated $\forall \bar{a}.C_1 \Rightarrow t_1$ with respect to the program logic P. In a type-passing translation scheme it suffices to check for "logical" subsumption. The target translation of $x = e_1$ is therefore simply $x = \Lambda\bar{a}.E_1$. Under a dictionary-passing scheme we would need a "constructive" subsumption check which must yield a proof c (i.e. coercion) to turn the target expression $\Lambda\bar{b}.E_1$ of type $\forall\bar{b}.C_1' \Rightarrow t_1'$ into a target expression $\Lambda\bar{a}.(\Lambda\bar{b}.E_1)(c\ \bar{a})$ of the expected type $\forall\bar{a}.C_1 \Rightarrow t_1$.

Rules (Inst) and (Advice) translate instance and advice declarations. The typecase statement is syntactic sugar for case. We call the third judgment to translate the advice and instance bodies into target expressions. Both rules are very similar which is no surprise given that we could explain advise in terms of type classes. As in case of let statements, we verify that the inferred type subsumes the annotated type. In case of instances, we check that the inferred type subsumes the declared type of the method where the type class parameters $\bar{a}$ are instantiated by $\overline{t_i}$.

For example, we can apply our translation scheme to the program in Figure 4 which yields the target program in Figure 6.

In a practical implementation, we might want to use the standard dictionary-passing scheme for the type class part and only use the type-passing scheme for translating the advice. In fact, we could use generalized algebraic data types [16] to encode the type-passing scheme to enable an integration of our translation scheme into the typed intermediate language of a compiler such as GHC.

5.5 Results

The following results are (mostly) immediate consequences of results found in [24, 19].

5.5.1 Type Soundness

We write $\eta \models \Gamma$ if $\eta(x) \in [\![\sigma_t]\!]_t$ for each $(x : \sigma_t) \in \Gamma$.

THEOREM 1 (Type Soundness). *Let $p \vdash E$ such that $\eta \models \Gamma_{init}$. Then, $[\![E]\!]\eta \neq \mathbf{W}$.*

The above results follows directly from Theorem 2 in [24]. In case we included functional dependencies in our description, we additionally require consistency [22] to maintain type soundness.

5.5.2 Type Inference

To obtain a decidable type inference algorithm, we will need algorithms to decide $P \models C$ and subsumption which boils down to deciding $P \wedge C' \models \exists\bar{a}.(C \wedge t = t')$. From [19], we know that $P \wedge C' \models \exists\bar{a}.(C \wedge t = t')$ can be rephrased as $P \wedge C' \wedge t = t' \models C$ which effectively means that under P, $C' \wedge t = t'$ entails C written $C' \wedge t = t' \supset C$. W.l.o.g. t and t' refer to variables assuming that we enrich the constraint language with type equations. We can safely "remove" these type equations via unification [17]. None of the CHRs contains type equations on the right-hand side. Hence, to decide subsumption it suffices to decide $P \models C_1 \supset C_2$ where C_1 and C_2 contain type class constraints. There is an implicit quantifier $\forall\bar{a}$ scoping over $C_1 \supset C_2$ where $\bar{a} = fv(C_1, C_2)$. We will leave this quantifier implicit. Notice that P is a closed formula. Hence, our task is to devise an algorithm to decide $P \models C_1 \supset C_2$ which will supply us with an algorithm to decide $P \models C$ as well.

In [19], we showed how to reduce $P \models C_1 \supset C_2$ to CHR solving. We apply CHRs on C_1 and $C_1 \wedge C_2$ and check whether we reach the same canonical normal form. The only slight complication here is that we use CHRs with guard constraints which were only briefly covered in [19]. For example, consider the translation of the program in Figure 1 where we assume that `insert` carries the type annotation `insert :: Ord a => a -> [a] -> [a]`. In the translation of `insert`, the subsumption check boils down to checking

$$P \models \mathit{Ord}\ a \supset \mathit{Advice}\ N_2\ (a \to [a] \to [a]) \quad (*)$$

where

$$P = \left\{ \begin{array}{l} \mathit{Advice}\ N_2\ (\mathit{Int} \to [\mathit{Int}] \to [\mathit{Int}]) \Longleftrightarrow \mathit{True}, \\ \mathit{Advice}\ N_2\ a \Longleftrightarrow a \neq (\mathit{Int} \to [\mathit{Int}] \to [\mathit{Int}]) \mid \mathit{True}, \\ \mathit{Ord}\ \mathit{Int} \Longleftrightarrow \mathit{True} \end{array} \right\}$$

We ignore here advice N_1 and include the CHR representing the Haskell Prelude `instance Ord Int`. The trouble is that none of the CHRs applies to $\mathit{Advice}\ N_2\ (a \to [a] \to [a])$. The first CHR does not apply because we use matching and not unification when firing CHRs. The second CHR does not apply because of the guard constraint. Although, logically the statement (*) clearly holds.

Our solution is to simply perform a case analysis. In essence, we perform solving by search. In case $a = \mathit{Int}$, we verify (*) by resolving $\mathit{Ord}\ a$ to True via the third CHR and $\mathit{Advice}\ N_2\ (a \to [a] \to [a])$ resolves to True via the first CHR. Hence, (*) holds for $a = \mathit{Int}$. In case $a \neq \mathit{Int}$, $\mathit{Advice}\ N_2\ (a \to [a] \to [a])$ resolves to True via the second CHR. In summary, we have verified (*) by case analysis.

We formalize this observation. First, we repeat the CHR operational semantics.

DEFINITION 5 (CHR Operational Semantics). *A CHR*

$$TC\ \bar{t} \iff \bar{t} \neq \overline{t'} \mid TC_1\ \overline{t_1}, ..., TC_n\ \overline{t_n}$$

applies to a constraint C if we find $TC\ \overline{t''} \in C$ such that $\phi(\bar{t}) = \overline{t''}$ and $\phi(\bar{t})$ and $\phi(\overline{t'})$ are not unifiable for some substitution ϕ. We assume that we rename CHRs before application to avoid name clashes. In such a situation, we write

$$C \rightarrowtail C - TC\ \overline{t''} \cup \{TC_1\ \phi(\overline{t_1}), ..., TC_n\ \phi(\overline{t_n})\}$$

to denote the constraint rewriting step using the above rule. We treat constraints as sets of type class constraints and write $C - tc$ to denote the constraint resulting from C where tc has been removed.

We write $C \rightarrowtail^* C'$ to denote exhaustive application of CHRs on initial *constraint C yielding the* final *constraint C' on which no further CHRs are applicable.*

The entailment checking algorithm is given in Figure 15. By construction, we know that Advice constraints only appear on the right-hand side of the entailment. In case (1), we can directly apply the first CHR which belongs to the advice declaration. Case (2) applies if t and t' are not unifiable. Then, we can directly apply the

```
entail(P | C_1 ⊃ C_2) =
  if ∃Advice N t' ∈ C &&
     {Advice N t ⟺ C, Advice N a ⟺ a ≠ t | True} ∈ P
  then if ∃φ.φ(t) = t' -- (1)
       then entail(P | C_1 ⊃ (C_2 − Advice N t' ∪ φ(C')))
       elseif ¬∃φ.φ(t) = φ(t') -- (2)
       then entail(P | C_1 ⊃ (C_2 − Advice N t'))
       else let φ be the mgu of t and t'; -- (3)
            entail(P | φ(C_1 ⊃ (C_2 − Advice N t' ∪ C')));
            entail(P | φ(C_1 ⊃ (C_2 − Advice N t')))
  else C_1 ↣* C_1'; -- (4)
       C_2 ↣* C_2';
       if C_1' = C_2' then return else abort
```

Figure 15. Entailment Checking Algorithm

second CHR which belongs to the "default" advice. In case (3), we build the most general unifier (mgu) among t and t' and perform a case analysis by considering the possibility that both CHRs are applicable. Case (4) is the "standard" case where use the entailment procedure from [19] and check whether the canonical normal forms of C_1 and $C_1 \wedge C_2$ are equivalent.

The important result is that the `entail` procedure retains all the nice properties we know from [19]. We say that p is a *complete and decidable* AOP Mini Haskell program if the set of CHRs resulting from instance declarations is terminating and the left-hand side of CHRs are non-overlapping. We need both properties to guarantee completeness and decidability for the "standard" case.

LEMMA 1. *Let p be a complete and decidable program such that $p \vdash \_, P, \_$ and C_1 and C_2 be two constraints. Then, we find the following results:*

1. *The procedure* `entail`$(P \,|\, C_1 \supset C_2)$ *is decidable.*
2. `entail`$(P \,|\, C_1 \supset C_2)$ *succeeds, iff $P \models C_1 \supset C_2$.*

For the above to hold, it is crucial that (by construction) there are no "cyclic" CHRs with guard constraints of the form

$$Foo\ [a] \iff a \neq Int \,|\, Foo\ a$$

We can therefore guarantee that in cases (1), (2) and (3) we will make progress and eventually reach the "standard" case (4). Also note that CHRs resulting from advice declarations are non-overlapping by construction because of the guard constraint.

We immediately obtain the following result.

THEOREM 2 (Type Inference). *Let p be a complete and decidable program. Then, type inference is decidable.*

We might hope to obtain a completeness result. However, there are well-known incompleteness problems in case of "nested" type annotations and type classes. We refer [23] for details. There is another source of incompleteness which is due to "ambiguous" programs. We will discuss this issue in the context of coherence which is our next topic.

5.6 Coherence

We would like to guarantee that regardless of the typing of the program the semantics of the target program is always the same. This property is known as coherence [1]. In the type class world, it is a well-known problem that we might lose coherence because of ambiguous programs. Think of the classic Show/Read example. The same problem arises in case of aspects in AOP Mini Haskell.

For example, consider

```
f :: [a] -> Int
f _ = 1
N@advice #f# :: [[Bool]] -> Int =  \x -> 2
main :: Bool
main = f undefined
```

Our pointcuts are type directed. However, we cannot unambiguously decide whether we apply advice `N` or the "default" advice. The problem becomes clear in the translation which shows that `(jp N f) undefined` has type `Bool` under the constraint `Advice N (a->Bool)`. Type variable `a` does not appear in the result type. Hence, we can freely choose `a`.

The solution employed for type classes is to reject ambiguous programs. We will follow this path for AOP Mini Haskell. Under this condition we can guarantee coherence as we will shortly see. A side-effect of rejecting ambiguous programs is that we lose completeness of type inference. Here are two (incomparable) annotations which make the program from above unambiguous.

```
main :: Bool
main = f (undefined::[[Bool]])
```

and

```
main :: Bool
main = f (undefined::[Int])
```

The conclusion is that in case we reject ambiguous programs we can only guarantee a *weak* form of completeness. That is, in case the principal derivation of a program is unambiguous, type inference will succeed. For the above example, type inference will fail because we reject ambiguous programs. Notice that the principal derivation for the above program is ambiguous. However, in the above we find that the program can be given two incomparable, unambiguous derivations. Hence, we cannot hope for a *strong* completeness result which guarantees that type inference with the addition of the unambiguity check succeeds if there exists an unambiguous derivation.

To state the coherence result concisely, we will first need to formally define unambiguity and a more general relation among type derivation. The following definitions can be found in similar form in [19].

We say $C, \Gamma \vdash e : t \leadsto E$ is *unambiguous* iff $fv(C) \subseteq fv(\Gamma, t)$.

We say a derivation $\mathcal{D}$ is unambiguous iff all judgments $C, \Gamma \vdash e : t \leadsto E$ in the derivation tree are unambiguous.

We say $C_1, \Gamma \vdash e : t_1$ is *more general* than $C_2, \Gamma \vdash e : t_2$ iff $P \vdash (\forall \bar{a}.C_1 \Rightarrow t_1) \leq (\forall \bar{b}.C_2 \Rightarrow t_2)$ where $\bar{a} = fv(C_1, t_1) - fv(\Gamma)$ and $\bar{b} = fv(C_2, t_2) - fv(\Gamma)$. In such a situation, we write $C_1, \Gamma \vdash e : t_1 \leq C_2, \Gamma \vdash e : t_2$.

We say a derivation $\mathcal{D}_1$ with final judgment $C_1, \Gamma \vdash e : t_1$ is *more general* than a derivation $\mathcal{D}_2$ with final judgment $C_2, \Gamma \vdash e : t_2$ iff for all judgments $C_1', \Gamma' \vdash e' : t_1'$ in $\mathcal{D}_1$ and $C_2', \Gamma' \vdash e' : t_2'$ in $\mathcal{D}_2$ which are at the same position in the derivation tree we have that $C_1', \Gamma' \vdash e' : t_1' \leq C_2', \Gamma' \vdash e' : t_2'$. Recall that the translation judgments for expressions are syntax-directed.

We say that a derivation $\mathcal{D}_1$ with final judgment $C_1, \Gamma \vdash e : t_1$ is *principal* iff there is no other more general derivation $\mathcal{D}_2$ with final judgment $C_2, \Gamma \vdash e : t_2$.

THEOREM 3 (Coherence). *Let p be a complete and decidable program such that the (1) principal derivation of p is unambiguous, (2) $p \vdash E_1$, (3) $p \vdash E_2$ and (4) $\eta \models \Gamma_{init}$. Then, $[\![E_1]\!]\eta = [\![E_2]\!]\eta$.*

The above follows directly from Theorem 15 in [19].

6. Conclusion and Related Work

There is a large amount of works on the semantics of aspect-oriented programming languages, for example consider [2, 13, 26, 27, 29] and the references therein. There have been only a few works [2, 14] which aim to integrate AOP into ML style languages. These impressive works substantially differ from ours. For instance, the work described in [2] supports first-class pointcuts and dynamic weaving whereas our pointcuts are second class and we employ static weaving. None of the previous works we are aware of of considers the integration of AOP and type classes. In some previous work, the second author [29, 28] gives a a static weaving scheme for a strongly typed functional AOP language via a type-directed translation process. However, there are no formal type inference and coherence results.

The main result of our work is that static weaving for strongly typed languages can be directly expressed in terms of type class resolution – the process of typing and translating type class programs. We could show that GHC type classes as of today can provide for a light-weight AOP extension of Haskell (Section 4). We critically rely on GHC's overlapping instance which imply a lazy and best-fit type class resolution strategy. We provided a number of programming examples in AOP Haskell light. [1] Programming in AOP Haskell light has the restriction that we are unable to advice polymorphic recursive functions. The restriction is due to the dictionary-passing translation scheme employed in GHC (Section 3.3). Therefore, we formalized a more principled and expressive AOP extension for a core fragment of Haskell, referred to as AOP Mini Haskell. Instead of overlapping instances we use guarded CHRs to represent advice and instead of a dictionary-passing scheme we use a type-passing scheme to translate AOP programs. Type class resolution is achieved via CHR solving by search. This is one of the main technical achievements of this work. We could state concise type soundness, type inference and coherence results for AOP Mini Haskell (Section 5). We believe that this system can serve as a foundational framework to study aspects and type classes.

In future work, we plan to investigate to what extent our results apply to other languages which support type classes. We also want to look into effect-full advice which we can represent via monads in Haskell. The study of more complex pointcuts is also an interesting topic for future work.

References

[1] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93(1):172–221, July 1991.

[2] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. PolyAML: a polymorphic aspect-oriented functional programming language. In *Proc. of ICFP'05*, pages 306–319. ACM Press, 2005.

[3] G. J. Duck, S. Peyton Jones, P. J. Stuckey, and M. Sulzmann. Sound and decidable type inference for functional dependencies. In *Proc. of ESOP'04*, volume 2986 of *LNCS*, pages 49–63. Springer-Verlag, 2004.

[4] T. Frühwirth. Constraint handling rules. In *Constraint Programming: Basics and Trends*, LNCS. Springer-Verlag, 1995.

[5] Glasgow haskell compiler home page. http://www.haskell.org/ghc/.

[6] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, 1996.

[7] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. of POPL'95*, pages 130–141. ACM Press, 1995.

[8] Fritz Henglein. Type inference with polymorphic recursion. *Transactions on Programming Languages and Systems*, 15(1):253–289, April 1993.

[1] We would like to point out that all examples from [29, 28] can be represented in the AOP extension of GHC. They are available via [20].

[9] M. P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Proc. of FPCA '93*, pages 52–61. ACM Press, 1993.

[10] M. P. Jones. Type classes with functional dependencies. In *Proc. of ESOP'00*, volume 1782 of *LNCS*. Springer-Verlag, 2000.

[11] S. Peyton Jones, M. P. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.

[12] S. Kaes. Parametric overloading in polymorphic programming languages. In *In Proc. of ESOP'88*, volume 300 of *LNCS*, pages 131–141. Springer-Verlag, 1988.

[13] Ralf Lämmel. A semantical approach to method-call interception. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 41–55. ACM Press, 2002.

[14] H. Masuhara, H. Tatsuzawa, and A. Yonezawa. Aspectual caml: an aspect-oriented functional language. In *Proc. of ICFP'05*, pages 320–330. ACM Press, 2005.

[15] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[16] S. Peyton Jones, D.Vytiniotis, G. Washburn, and S. Weirich. Simple unification-based type inference for GADTs. In *Proc. of ICFP'06*. ACM Press, 2006.

[17] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12:23–41, 1965.

[18] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Proc. of the ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM Press, 2002.

[19] P. J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1–54, 2005.

[20] M. Sulzmann. AOP Haskell light: Aspect-oriented programming with type classes. http://www.comp.nus.edu.sg/˜sulzmann/aophaskell.

[21] M. Sulzmann, M. Chakravarty, , S. Peyton Jones, and K. Donnelly. System F with type equality coercions. http://www.comp.nus.edu.sg/˜sulzmann, July 2006.

[22] M. Sulzmann, G. J. Duck, S. Peyton Jones, and P. J. Stuckey. Understanding functional dependencies via constraint handling rules. *Journal of Functional Programming*, 2006. To appear.

[23] M. Sulzmann, T. Schrijvers, and P. J. Stuckey. Principal type inference for GHC-style multi-parameter type classes. In *Proc. of APLAS'06*, 2006. To appear.

[24] S. R. Thatte. Semantics of type classes revisited. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 208–219. ACM Press, 1994.

[25] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. of POPL'89*, pages 60–76. ACM Press, 1989.

[26] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *Proc. of ICFP'03*, pages 127–139. ACM Press, 2003.

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

[28] M. Wang, K. Chen, and S.C. Khoo. On the pursuit of staticness and coherence. In *FOAL '06: Foundations of Aspect-Oriented Languages*, 2006.

[29] M. Wang, K. Chen, and S.C. Khoo. Type-directed weaving of aspects for higher-order functional languages. In *Proc. of PEPM '06: Workshop on Partial Evaluation and Program Manipulation*, pages 78–87. ACM Press, 2006.

[30] G. Washburn and S. Weirich. Good advice for type-directed programming: Aspect-oriented programming and extensible generic functions. In *Proc. of the 2006 Workshop on Generic Programming (WGP'06)*, pages 33–44. ACM Press, 2006.