Citation for published version


DOI

Link to record in KAR

http://kar.kent.ac.uk/47468/

Document Version

Author's Accepted Manuscript

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version. Users are advised to check http://kar.kent.ac.uk for the status of the paper. Users should always cite the published version of record.

Enquiries
For any further enquiries regarding the licence status of this document, please contact: researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at http://kar.kent.ac.uk/contact.html
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 classes. 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.
\[
\begin{align*}
\text{class Eq a where} & \quad (==) :: a \rightarrow a \rightarrow \text{Bool} \\
\text{instance Eq Int where} & \quad (==) = \text{primIntEq} \quad \text{-- (I1)} \\
\text{instance Eq a => Eq [a] where} & \quad (==) [] [] \Rightarrow \text{True} \quad \text{-- (I2)} \\
& \quad (==) (x:xs) (y:ys) \Rightarrow (x==y) \wedge (xs==ys) \quad \text{-- (L)} \\
& \quad (==) _ _ = \text{False} \\
\end{align*}
\]

The class declaration in the first line states that every type a in the class Eq has an equality function ==. Instance (I1) shows that Int is in Eq. We assume that \text{primIntEq} is the (primitive) equality function among Ints. 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 a 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.
\[
\begin{align*}
\text{class Eq a => Ord a where} & \quad (<) :: a \rightarrow a \rightarrow \text{Bool} \quad \text{-- (S1)} \\
\text{instance Ord Int where} & \quad \ldots \quad \text{-- (I3)} \\
\text{instance Ord a => Ord [a] where} & \quad \ldots \quad \text{-- (I4)} \\
\end{align*}
\]

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) &&
       eq _ _ = False
   in eq


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


Figure 1. AOP Haskell Example

3. The Key Ideas
3.1 AOP Haskell
AOP Haskell extends the Haskell syntax [15] by supporting top-level aspect definitions of the form

\[
\text{NAbstract #a1, \ldots, fn# :: C => t = e}
\]

where B is a distinct label attached to each advice and the pointcut \( f_1 \ldots f_n \) refers to a set of (possibly overloaded) functions. Commonly, we refer to \( f_i \)'s as joinpoints. Notice that our pointcuts are type-directed. Each pointcut has a type annotation \( C \Rightarrow t \) which follows the Haskell syntax. We refer to \( C \Rightarrow t \) as the pointcut type. We will apply the advice if the type of a joinpoint \( f_i \) is an instance of \( t \) such that constraints \( C \) are satisfied. The advice body 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 \( \text{insert} \) takes a sorted list as an input argument and returns a sorted list as the result.

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 \( \text{sort} \) which sorts a list of values. The sort functions assumes the overloaded comparison operator \( <= \) which is part of the \( \text{Ord} \) class. Hence, we find the pointcut type \( \text{Ord a} \Rightarrow a \rightarrow [a] \rightarrow [a] \). The keyword proceed indicates to continue with the normal evaluation. That is, we continue with the call \( \text{insert} \) to \( [a] \). The second aspect definition provides for a more efficient implementation in case we call \( \text{insert} \) on list of \( \text{Ints} \). We assume that only non-negative numbers are sorted which implies that 0 is the smallest element appearing in a list of \( \text{Ints} \). 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 \( \text{insert} \). In case we call \( \text{insert} \) on list of \( \text{Ints} \) 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 \( = \) which is part of the \( \text{Eq} \) class. Hence, we find the pointcut type \( \text{Eq a} \Rightarrow a \rightarrow [a] \rightarrow [a] \). The keyword proceed indicates to continue with the normal evaluation. That is, we continue with the call \( \text{insert} \) to \( [a] \). The second aspect definition provides for a more efficient implementation in case we call \( \text{insert} \) on list of \( \text{Ints} \). We assume that only non-negative numbers are sorted which implies that 0 is the smallest element appearing in a list of \( \text{Ints} \). 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 \( \text{insert} \). In case we call \( \text{insert} \) on list of \( \text{Ints} \) 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 \( = \) which is part of the \( \text{Eq} \) class. Hence, we find the pointcut type \( \text{Eq a} \Rightarrow a \rightarrow [a] \rightarrow [a] \). The keyword proceed indicates to continue with the normal evaluation. That is, we continue with the call \( \text{insert} \) to \( [a] \). The second aspect definition provides for a more efficient implementation in case we call \( \text{insert} \) on list of \( \text{Ints} \). We assume that only non-negative numbers are sorted which implies that 0 is the smallest element appearing in a list of \( \text{Ints} \). 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 \( \text{insert} \). In case we call \( \text{insert} \) on list of \( \text{Ints} \) 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 \( = \) which is part of the \( \text{Eq} \) class. Hence, we find the pointcut type \( \text{Eq a} \Rightarrow a \rightarrow [a] \rightarrow [a] \). The keyword proceed indicates to continue with the normal evaluation. That is, we continue with the call \( \text{insert} \) to \( [a] \). The second aspect definition provides for a more efficient implementation in case we call \( \text{insert} \) on list of \( \text{Ints} \). We assume that only non-negative numbers are sorted which implies that 0 is the smallest element appearing in a list of \( \text{Ints} \). 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 \( \text{insert} \). In case we call \( \text{insert} \) on list of \( \text{Ints} \) 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 \( = \) which is part of the \( \text{Eq} \) class. Hence, we find the pointcut type \( \text{Eq a} \Rightarrow a \rightarrow [a] \rightarrow [a] \). The keyword proceed indicates to continue with the normal evaluation. That is, we continue with the call \( \text{insert} \) to \( [a] \). The second aspect definition provides for a more efficient implementation in case we call \( \text{insert} \) on list of \( \text{Ints} \). We assume that only non-negative numbers are sorted which implies that 0 is the smallest element appearing in a list of \( \text{Ints} \). 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 \( \text{insert} \). In case we call \( \text{insert} \) on list of \( \text{Ints} \) 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 \( = \) which is part of the \( \text{Eq} \) class. Hence, we find the pointcut type \( \text{Eq a} \Rightarrow a \rightarrow [a] \rightarrow [a] \). The keyword proceed indicates to continue with the normal evaluation. That is, we continue with the call \( \text{insert} \) to \( [a] \). The second aspect definition provides for a more efficient implementation in case we call \( \text{insert} \) on list of \( \text{Ints} \). We assume that only non-negative numbers are sorted which implies that 0 is the smallest element appearing in a list of \( \text{Ints} \). 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 \( \text{insert} \). In case we call \( \text{insert} \) on list of \( \text{Ints} \) 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 \( = \) which is part of the \( \text{Eq} \) class. Hence, we find the pointcut type \( \text{Eq a} \Rightarrow a \rightarrow [a] \rightarrow [a] \). The keyword proceed indicates to continue with the normal evaluation. That is, we continue with the call \( \text{insert} \) to \( [a] \). The second aspect definition provides for a more efficient implementation in case we call \( \text{insert} \) on list of \( \text{Ints} \). We assume that only non-negative numbers are sorted which implies that 0 is the smallest element appearing in a list of \( \text{Ints} \). 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 \( \text{insert} \). In case we call \( \text{insert} \) on list of \( \text{Ints} \) 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 \( = \) which is part of the \( \text{Eq} \) class. Hence, we find the pointcut type \( \text{Eq a} \Rightarrow a \rightarrow [a] \rightarrow [a] \). The keyword proceed indicates to continue with the normal evaluation. That is, we continue with the call \( \text{insert} \) to \( [a] \). The second aspect definition provides for a more efficient implementation in case we call \( \text{insert} \) on list of \( \text{Ints} \). We assume that only non-negative numbers are sorted which implies that 0 is the smallest element appearing in a list of \( \text{Ints} \). 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 \( \text{insert} \). In case we call \( \text{insert} \) on list of \( \text{Ints} \) we apply both advice bodies in no specific order unless otherwise stated. For all other cases, we only apply the first advice.
We employ type class instances to represent advice. We use a syntactic pre-processor to instrument joinpoints with insertionSort [] = []
is more restrictive but allows us to give stronger static guarantees.

flexible approach. For example, aspects can be added and removed
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

\[ \text{joinpoint } N1 (\text{joinpoint } N2 \text{ insert}) \]

We assume here the following order among advice: \( N2 \leq N1 \). 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 \( C \Rightarrow \ldots \) 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.

\[ \text{insert } :: \forall a. \]

\[ \text{insertionSort } :: \forall a. \]

\[ \text{isSorted } :: \forall a. \]

\[ \text{Ord } a => \text{Advice } N1 (a->[a]->[a]) \]

\[ \text{instance } (I1') \]

\[ \text{default case} \]

**Figure 2. GHC Haskell Translation of Figure 1**

The challenge we face is how to intercept calls to joinpoints and re-direct 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 syntaxic 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

\[ \text{joinpoint } N1 (\text{joinpoint } N2 \text{ insert}) \]

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.
Advice N2 (\(a \rightarrow [a]\)) and applying the default instance (I2') on by applying instance (I1) on

arising from

\[\text{joinpoint } N1 \ (a \rightarrow [a] \rightarrow [a])\]

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

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

\[\text{insert } :: \text{Ord } a \Rightarrow a \rightarrow [a] \rightarrow [a]\]

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 decidability [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

calls to \(f\) on list of list of \(\text{Bool}\)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.

\[f :: \text{Advice } N \ a \Rightarrow [a] \rightarrow \text{Bool}\]

\[f \ [] = \text{True}\]

\[f \ (x:xs) = (\text{joinpoint } N \ f) \ [xs]\]

The call to \(f\) in the function body gives rise to \(\text{Advice } N \ [a]\) whereas the annotation only supplies \(\text{Advice } N \ a\). 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 \(\text{Bool}\)s. Then, the default advice applies and we proceed with \(f\)'s evaluation. Subsequently, we will call \(f\) on a list of list of \(\text{Bool}\)s. Recall that \(f\) is a polymorphic recursive function. Now, we wish that the advice \(N\) applies to terminate the evaluation with result \(\text{False}\). 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.

3.3 Short-comings using GHC Style Type Classes

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

\[\text{insert } :: \text{Ord } a \Rightarrow a \rightarrow [a] \rightarrow [a]\]

\[\text{insertSort } :: \text{Ord } a \Rightarrow a \rightarrow [a] \rightarrow [a]\]

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

\[\text{Advice } N1 \ (a \rightarrow [a] \rightarrow [a]),\]

\[\text{Advice } N2 \ (a \rightarrow [a] \rightarrow [a])\]

arising from

\[\text{joinpoint } N1 \ (\text{joinpoint } N2 \ \text{insert})\]

\[\text{by applying instance (I1) on Advice } N1 \ (a \rightarrow [a] \rightarrow [a]) \text{ and applying the default instance (I2') on Advice } N2 \ (a \rightarrow [a] \rightarrow [a]).\]

Hence, will never apply the advise N2, even if we call insert on list of Ints.

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

\[\text{insert } :: \text{Ord } a \Rightarrow a \rightarrow [a] \rightarrow [a]\]

\[\text{Advice } N1 \ (a \rightarrow [a] \rightarrow [a]),\]

\[\text{Advice } N2 \ (a \rightarrow [a] \rightarrow [a]), \text{Ord } a \Rightarrow a \rightarrow [a] \rightarrow [a]\]

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 decidability [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

\[f :: [a] \rightarrow \text{Bool}\]

\[f \ [] = \text{True}\]

\[f \ (x:xs) = (\text{joinpoint } N \ f) \ [xs]\]

calls to \(f\) on list of list of \(\text{Bool}\)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.

\[f :: \text{Advice } N \ a \Rightarrow [a] \rightarrow \text{Bool}\]

\[f \ [] = \text{True}\]

\[f \ (x:xs) = (\text{joinpoint } N \ f) \ [xs]\]

The call to \(f\) in the function body gives rise to \(\text{Advice } N \ [a]\) whereas the annotation only supplies \(\text{Advice } N \ a\). 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 \(\text{Bool}\)s. Then, the default advice applies and we proceed with \(f\)'s evaluation. Subsequently, we will call \(f\) on a list of list of \(\text{Bool}\)s. Recall that \(f\) is a polymorphic recursive function. Now, we wish that the advice \(N\) applies to terminate the evaluation with result \(\text{False}\). 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,
We demand that
- Type soundness and type inference for AOP Haskell light
- For each function
- Each AOP Haskell statement
- fallow-overlapping-instances
- Each AOP Haskell statement
- We add the class declaration

Proceed

We omit to give the syntactic description of Haskell programs
which can be found elsewhere [15]. We assume that type anno-
notations, advice or

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

tions 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 transfor-
mation steps on \( p \) to obtain the program \( p' \).

Advice class: We add the class declaration

\begin{align*}
\text{class} \quad & \text{Advice n t where} \\
& \text{joinpoint :: n -> t -> t} \\
& \text{joinpoint _ = id}
\end{align*}

Advice bodies: Each AOP Haskell statement

\begin{align*}
\text{N@advice \#f1,...,fn# :: C => t = e}
\end{align*}

is replaced by

\begin{align*}
\text{data N = N} \\
\text{instance C => Advice N t where} \\
& \text{joinpoint _ f = e'} \\
\text{instance Advice N a -- default case} \\
& \text{where e' results from e by substituting proceed by the fresh} \\
& \text{name f.}
\end{align*}

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

\begin{align*}
& \text{joinpoint N1 (...) (joinpoint Nm f)...)} \\
& \text{being careful to avoid name conflicts in case of lambda-bound} \\
& \text{function names. We assume that the order among advice is as} \\
& \text{follows: Nm \leq ... \leq N1.}
\end{align*}

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

\begin{align*}
f :: \text{Int} \\
f = 1
\end{align*}

\begin{align*}
\text{N@advice \#f\# :: Bool = True}
\end{align*}

where the advice definition \( N \) is clearly useless. We may want to re-
ject such useless definitions by adding the following transformation
step to Definition 2.

Useful Advice: Each AOP Haskell statement

\begin{align*}
\text{N@advice \#f1,...,fn# :: C => t = e}
\end{align*}

generates

\begin{align*}
f1' :: C => t \\
f1' = f \\
... \\
fn' :: C => t \\
fn' = fn
\end{align*}

in \( p' \) where \( f1', ..., f_n' \) 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 \( C \Rightarrow t \).

In case of our above example, we generate

\begin{align*}
f' :: \text{Bool} \\
f' = f
\end{align*}

which is ill-typed. Hence, we reject the useless advice \( N \).
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. 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 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).

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

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.
For our purposes, CHRs are of the form
\[ \text{TC } T \iff \exists \bar{t} \in \mathcal{F} \text{ such that } (\text{TC } T \equiv \exists \bar{e} (\text{TC } t_1 \land \ldots \land \text{TC } t_n)) \]

Logically, we interpret the above as the first-order formula:
\[ \forall a.((\exists \bar{t} \neq \bar{t}) \supset (\text{TC } T \equiv \exists \bar{e} (\text{TC } t_1 \land \ldots \land \text{TC } t_n))) \]

We formalize the syntax and semantics of CHRs. We assume that \( f(y) \) computes the free variables of some object \( a \). 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.
where \( \bar{a} = f\bar{v}(\bar{v}), \bar{b} = f\bar{v}(\bar{v}) \) and \( \bar{v} = f\bar{v}(\bar{v}_1, \ldots, \bar{v}_n) \). The above formula simplifies to \( \forall \bar{a}.(T \bar{C} \triangleright \bar{a}) \)
\( = (T \bar{C}_1 \triangleright \bar{a}_1 \land \ldots \land T \bar{C}_n \triangleright \bar{a}_n) \).

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 \( \forall \bar{a}.C' \Rightarrow t' \) be two types. We define

\[
P \triangleright (\forall \bar{a}.C \Rightarrow t) \leq (\forall \bar{a}.C' \Rightarrow t')
\]

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

The statement \( P \land C' \models \exists \bar{a}.(C \land t = t') \) holds if for any model \( M \) of \( P \land C' \) (in the first-order sense) we find that \( \exists \bar{a}.(C \land t = t') \) 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. We will write letrec \( x = E_1 \in E_2 \) as syntactic sugar for let \( x = (\mathbf{rec} \ f \ E_1) \in E_2 \) where \( f \) is a fresh identifier. Multiple binding groups let \( x_1 = \bar{E}_1, \ldots, x_n = \bar{E}_n \in E \) can also be desugared into letrec \( x = E \in E' \) 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 TVale. 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 \( \forall \rightarrow \) is the continuous function space. In general, we leave injection and projection operators for sums implicit. The value \( W \) is the error element, \( K \) is the set of value constructors and \( \text{fix} \) refers to the fix-point operator. In the rule for case expressions, we write \( \eta \vartheta(v) = \eta_v(v_1) \) to denote that the value \( \eta \vartheta(v) \) is matched against \( v_1 \) for some (local) value binding \( \eta_v \). We write \( \eta \vartheta \eta_v \) to denote composition of value bindings.

In the second semantic (type) equation from the bottom, we write \( \Gamma_{\text{target}} \vdash K : \mu_1 \rightarrow \cdots \rightarrow \mu_n \rightarrow T \mu_1 \cdots \mu_m \) 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: \( C, \Gamma \vdash e : t \leadsto E \).

### Figure 13. Target Language

4. Instanciation: \( \Gamma \vdash \lambda \mathbf{tt} \sim j p = E \).
5. Advice: \( \Gamma \vdash \lambda \mathbf{tt}_{PC} \sim m = E \).

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

The first judgment drives the translation process. In the premise of rule Prog, we call the second judgment to collect the set \( \Gamma \) 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 \( N \) (singleton) type \( N \) in the initial environment \( \Gamma_{\text{init}} \).

Then, we call the fourth and fifth judgment to translate the advice and instances. We write \( \mathbf{tt}_{PC} \) to denote a sequence of instance declarations which refer to type class \( TC \) in their instance "head". The result is sequence of binding groups \( jp = E', m_1 = \ldots = \ldots \).
Figure 14. Mini AOP Haskell Translation Scheme
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 \( I/f/\bar{a} \) to denote a substitution mapping variables \( a_i \) to types \( t_i \). Rule (Var-IP) 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 \( \{ \langle f', N \rangle \in J \mid f = f' \} \). 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 \( C \implies t \). 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 \( \forall b.C_1 \implies t_1 \) subsumes the annotated type \( \forall a.C_1 \implies t_1 \) 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 \( \Lambda b.E_1 \) of type \( \forall b.C_1 \implies t_1 \) into a target expression \( \Lambda \bar{a} (\Lambda b.E_1)c \) of the expected type \( \forall a.C_1 \implies t_1 \).

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 here 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

\[
P \models \text{Ord } a \circ \text{Advice } N_2 (a \to a) \to [a] \tag{*}
\]

where

\[
P = \{ \text{Advice } N_2 (\text{Int } \to [\text{Int } \to \text{Int}]) \iff \text{True}.
\]

We ignore here advice \( N_i \) and include the CHR representing the Haskell Prelude instance \( \text{Ord } \text{Int} \). The trouble is that none of the CHRs applies to \( \text{Advice } N_2 (a \to a) \to [a] \). 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 \( a = \text{Int} \), we verify (*) by resolving \( \text{Ord } a \to \text{True} \) via the third CHR and Advice \( N_2 (a \to [a]) \) resolves to True via the first CHR. Hence, (*) holds for \( a = \text{Int} \). In case \( a \neq \text{Int} \), Advice \( N_2 (a \to [a]) \) 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

\[
TC \quad \iff \quad \overrightarrow{\tau} \not\in TC \quad TC_1; TC_2; \ldots; TC_n \quad TC_n
\]

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

\[
C \vdash C - TC \overrightarrow{\tau} \cup \{ TC_1; TC_2; \ldots; TC_n \}
\]

technically amounts to a denotation of 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 \( C \vdash C' \) 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 \vdash C_1 \supset C_2) = 
\begin{align*}
\text{if } \exists \text{Advice } N \ t' \in C & \land \land \\
\{ \text{Advice } N \ t \iff C, \text{Advice } N \ a \iff a \neq t \land \text{True} \} \in P \\
\text{then if } \exists \phi(t) = t' & \rightarrow (1) \\
\text{then entail}(P \vdash C_2 - \text{Advice } N \ t' \cup \phi(C')) \\
\text{elseif } \neg \exists \phi(t) = \phi(t') & \rightarrow (2) \\
\text{then entail}(P \vdash C_2 - \text{Advice } N \ t') \\
\text{else let } \phi \text{ be the mgu of } t \text{ and } t' & \rightarrow (3) \\
\text{entail}(P \vdash \phi(C_1 \supset (C_2 - \text{Advice } N \ t' \cup C'))); \\
\text{entail}(P \vdash \phi(C_1 \supset (C_2 - \text{Advice } N \ t')))
\end{align*}
\begin{align*}
\text{else } C_1 & \mapsto C_1'; \rightarrow (4) \\
C_2 & \mapsto C_2'; \\
\text{if } C_1' = C_2' \text{ then return else abort}
\end{align*}

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 \( C_1 \land C_2 \) 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(\( P \vdash C_1 \supset C_2 \)) is decidable.
2. entail(\( P \vdash C_1 \supset C_2 \)) succeeds, iff \( P \vdash C_1 \supset C_2 \).

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

\[ \text{Foo } [a] \iff a \neq \text{Int]\text{Foo } a \]

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

\[ f :: [a] \rightarrow \text{Int} \]

\[ \text{f } _ = 1 \]

\[ \text{N@advice #} \# :: [[\text{Bool}]] \rightarrow \text{Int} = \text{x } \rightarrow 2 \]

main :: Bool

main = f undefined

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 \( C, \Gamma \vdash e : t \sim E \) is unambiguous iff \( fn(C) \subseteq fn(\Gamma, t) \).

We say a derivation \( D \) is unambiguous iff all judgments \( C, \Gamma \vdash e : t \sim E \) 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 \( P \vdash (\forall a. C_1 \Rightarrow t_1) \leq (\forall b. C_2 \Rightarrow t_2) \) where \( \hat{a} = fn(C_1, t_1) - fn(\Gamma) \) and \( \hat{b} = fn(C_2, t_2) - fn(\Gamma) \). 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 \( D_1 \) with final judgment \( C_1, \Gamma \vdash e : t_1 \) is more general than a derivation \( D_2 \) with final judgment \( C_2, \Gamma \vdash e : t_2 \) iff for all judgments \( C_1', \Gamma' \vdash e' : t_1' \) in \( D_1 \) and \( C_2', \Gamma' \vdash e' : t_2' \) in \( D_2 \) 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 \( D_1 \) with final judgment \( C_1, \Gamma \vdash e : t_1 \) is principal iff there is no other more general derivation \( D_2 \) 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) \( p \vdash E_1 \Rightarrow E_2 \). Then, \( [E_1]p = [E_2]p \).

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, 2006/10/3]
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 considers the integration of AOP and type classes. In some previous work, the second author [29, 28] gives 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 overloading instance which imply a lazy and best-fit type class resolution strategy. We provided a number of programming examples in AOP Haskell light. 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 overloading 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 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].