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 the concept of type classes, a type system which supports the interception of events at run-time. The essential functionality provided by 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 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 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 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:

```
instance Eq a => Eq [a] where -- (I2)
   (==) [] [] = True
   (==) (x:xs) (y:ys) = (x==y) && (xs==ys) -- (L)
   (==) _ _ = False
```
We will apply the advice if the type of a joinpoint \( C \Rightarrow t \) follows the Haskell syntax. We refer to \( C \Rightarrow t \) monly, we refer to \( f_1 \). In the above, we only use single-parameter type classes. Other addi-
tional type class features include functional dependency [10], constructor [9] and multi-parameter [11] type classes. For the transla-
tion 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

\[
N@advice \ #1, \ldots, \ #n \ #:: \ C \Rightarrow t = e
\]

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. Common-
ly, we refer to \( f_1 \) as \emph{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 \emph{pointcut type}. We will apply the advice if the type of a joinpoint \( f_1 \) is an instance of \( t \) such that constraints \( C \) are satisfied. The advice body \( e \) fol-
lows the Haskell syntax for expressions with the addition of a new keyword \emph{proceed} to indicate continuation of the normal evalu-
ation 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 pro-
vide the implementation of an insertion sort algorithm where ele-
ments 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 \emph{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 \emph{sort} which sorts a list of values. The \emph{sort} functions assumes the overloaded comparison operator \( < \) which is part of the \emph{Ord} class. Hence, we find the pointcut type \( \text{Ord} a \Rightarrow a \rightarrow [a] \rightarrow [a] \). The keyword \emph{proceed} indicates to continue with the normal evaluation. That is, we continue with the call \( \text{insert} \ x \ y : y : \text{ys} \) if \( x \leq y \). The second aspect definition provides for a more efficient implementation in case we call \emph{insert} on list of \emph{Int}s. We assume that only non-negative numbers are sorted which implies that \( 0 \) is the smallest element appearing in a list of \emph{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 \emph{insert}. In case we call \emph{insert} on list of \emph{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 \( =\) which is part of the \emph{Ord} class. Hence, we find the pointcut type \( \text{Eq} a \Rightarrow a \rightarrow [a] \rightarrow [a] \Rightarrow [a] \). The keyword \emph{proceed} indicates to continue with the normal evaluation. That is, we continue with the call \( \text{insert} \ x \ ys \) which is part of the \emph{Ord} class. Hence, we find the pointcut type \( \text{Ord} a \Rightarrow a \rightarrow [a] \rightarrow [a] \Rightarrow [a] \Rightarrow [a] \). The keyword \emph{proceed} indicates to continue with the normal evaluation. That is, we continue with the call \( \text{insert} \ x \ ys \) if \( x \leq 0 \). Hence, there is no need to mention the \emph{Eq} class in the pointcut type of the advice definition. Besides ordinary function, we can advise

\begin{itemize}
  \item overloading functions,
  \item polymorphic recursive functions, and
  \item functions appearing in advice and instance bodies.
\end{itemize}

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 \emph{source} AOP Haskell program is described by the resulting \emph{target} Haskell program.
transformation step requires to traverse the abstract syntax tree. We assume here the following order among advice:

\[ N_2 \leq N_1 \]

That is, we first apply the advice \( N_1 \) before applying advice \( N_2 \). 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 \( N_1 \). 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:

\[
\text{insert :: forall } a. \\
\quad (\text{Advice } N_1 (a \to [a] \to [a]), \text{Ord } a) \Rightarrow \\
\quad a \to [a] \to [a] \\
\text{insertSort :: forall } a. \\
\quad (\text{Advice } N_1 (a \to [a] \to [a]), \text{Ord } a) \Rightarrow \\
\quad a \to [a] \\
\text{isSorted xs = (sort xs) == xs} \\
\text{isSorted xs = (sort xs) == xs}
\]

Figure 2. GHC Haskell Transformation 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 runtime. 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

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

We assume here the following order among advice: \( N_2 \leq N_1 \). That is, we first apply the advice \( N_1 \) before applying advice \( N_2 \). This transformation step requires to traverse the abstract syntax tree
Figure 3. AOP Haskell Typing and Translation Scheme

```
AOP Haskell
  ↓
Haskell+Type Classes
  ↓
  * turn advice into instances
  * instrument joinpoints
  * type class resolution
  * further compilation steps
```

Figure 4. Advising Polymorphic Recursive Functions

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

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

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 (I’₂) on `Advice N2 (a -> [a] -> [a])`. Hence, we 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

```
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 calls to `f` on list of list of `Bools` 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) [x]
```

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 `Bools`. Then, the default advice applies and we proceed with `f`’s evaluation. Subsequently, we will call `f` on a list of list of `Bools`. 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,
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 produce 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 \( N1,...,N_m \) 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: \( N_m \leq ... \leq N_1 \).
```

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 subsection.

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 \text{xs \ acc} = \text{accF} (\text{tail \ xs}) (\text{head \ xs : acc})
reverse :: [a] -> [a] -> [a]
reverse \text{xs} = \text{accF} \text{xs} []
append :: [a] -> [a] -> [a]
append \text{xs \ ys} = \text{accF} \text{xs} \text{ys}

\text{N@advice f}{\text{accF}} :: [a] -> [a] -> [a] = \text{xs -> \text{acc -> case \ xs of}}
\text{[] -> acc}
\text{. -> proceed \text{xs \ acc}}

\text{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 \text{[]} = \text[x]
insert x (y:ys) | x <= y = x:y:ys
| otherwise = y : (insert x ys)
test x xs = elem x xs
empty = []

\text{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 \text{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 \text{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 \text{Collects} c e | c -> e to enforce that the collection type \text{c} uniquely determines the element type \text{e}. We use the same aspect definitions from earlier on to advise function \text{insertionSort} and the now overloaded function \text{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 \text{insert} throughout the entire program, we will need to place the entire code into one single module. Let us assume we replace the statement \text{import CollectsLib} in Figure 9 by the code in Figure 8 (dropping the statement \text{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 \text{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

... instance \text{(Advice N1 (a->[a]->[a]),}
\text{Advice N2 (a->[a]->[a]),}
\text{Ord a) => Collects [a] a where}
insert x \text{[]} = \text[x]
insert x (y:ys) | x <= y = x:y:ys
| otherwise = y : (insert x ys)
test x xs = elem x xs
empty = []

...
Then, we develop some technical machinery necessary to concisely state some formal results. 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. 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 automate the rewriting of annotations by integrating the translation scheme in Definition 2 with the GHC type inferencer. However, we do not have to perform any manual rewriting. Of course, we could apply. Hence, in the type class translation we use the default instance Advice N1 a

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 \( a \) as a short-hand to denote a sequence of objects \( o_1, ..., o_n \). We assume a distinct type class \( \text{True} \) representing the always true constraint. We write \( \forall \alpha . t \) as a short-hand for \( \forall \alpha . \text{True} \Rightarrow t \). 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 head : \( \forall a . [a] \rightarrow a \) and tail : \( \forall a . [a] \rightarrow [a] \) which are recorded in some initial environment \( \Gamma_{\text{init}} \).

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 (CHR) [4]. CHRs serve as a meta specification language to reason about type class relations.

For example, the instance declaration from Figure 8

\[
\text{instance Ord} \ a \Rightarrow \text{Collecta} [a] \ a
\]

translates to the CHR

\[
\text{Collecta} [a] \ a \Rightarrow \text{Ord} \ a
\]

Logically, the symbol \( \Leftrightarrow \) 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

\[
\text{N1 advice } \#\text{insert}\# :: \text{Ord} \ a \Rightarrow a \rightarrow [a] \rightarrow [a] = \ldots
\]

from Figure 1 translates to the overlapping instances

\[
\text{instance Ord} \ a \Rightarrow \text{Advice} \ N1 \ (a \rightarrow [a] \rightarrow [a])
\]

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.

Advise N1 (a->[a]->[a]) <= Ord a

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 \( f(a) \) 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.

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

\[
TC \ T \Leftrightarrow \ T \neq \emptyset \mid TC_1 \ T_1 \land \ldots \land TC_n \ T_n
\]

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

\[
\forall \bar{x}. (\exists T \neq \emptyset) \supset (TC \ T \Leftrightarrow \exists \bar{e}.(TC_1 \ T_1 \land \ldots \land TC_n \ T_n)))
\]
where $\tilde{a} = \mathit{fv}(\tilde{t}), \tilde{b} = \mathit{fv}(\tilde{T}) - \tilde{a}$ and $\tilde{e} = \mathit{fv}(\tilde{T}_1, \ldots, \tilde{T}_n) - \tilde{a}$. The above formula simplifies to $\forall \tilde{a}.(\mathit{Tc} \; \tilde{T} \rightarrow \exists \tilde{e}.(\mathit{Tc}_1 \; \tilde{T}_1 \wedge \ldots \wedge \mathit{Tc}_n \; \tilde{T}_n))$ 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 $\forall \tilde{a}.C \Rightarrow t$ and $\forall \tilde{a}'.C' \Rightarrow t'$ be two types. We define

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

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

The statement $P \land C' \models \exists \tilde{a}.(C \land t = t')$ holds if for any model $M$ of $P \land C'$ in the (first-order sense) we find that $\exists \tilde{a}.(C \land t = t')$ 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. In the translation, we will write letrec $x = E_1$ in $E_2$ as syntactic sugar for $let x = \mathit{rec} f \in \{ f \in \mathit{Var} \mid x \in \{m_1 \}, \ldots, x \in \{m_2 \} \} \Rightarrow f \in \{m_3 \}$.

We use $\mathit{Var}$ for the set of (singleton) type class declarations. We directly translate advice declarations into intermediate step where we first translate advice declarations into type class constructs and then translating type classes, we immediately translate AOP and type class constructs to the target language.

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.

4. Instances: $\Gamma \vdash \mathit{adm} \rightsquigarrow \mathit{jp} = E$.

5. Advice: $\Gamma \vdash \mathit{adm} \rightsquigarrow m = E$.

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 $\Gamma$ 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 $\mathit{Advice}$ 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 $\Gamma_{init}$.

Then, we call the fourth and fifth judgment to translate the advice and instances. We write $\mathit{adm}_{\mathit{TC}} \rightsquigarrow nb$ to denote a sequence of instance declarations which refer to type class $\mathit{TC}$ in their instance “head”. The result is sequence of binding groups $\mathit{jp} = E$, $m_1 = E$.
Mini AOP Haskell Translation Scheme
to deciding $P \land C' \models \exists \bar{a}.(C \land t' = t')$. From [19], we know that $P \land C' \models \exists \bar{a}.(C \land t = t')$ can be rephrased as $P \land C' \land t = t' \models C$ which effectively means that under $P$, $C' \land t = t'$ entails $C$ written $\Lambda^{C'} \land t = t' \models C$. 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 $P \models \exists \bar{a}.C_1 \land C_2$ where $C_1$ and $C_2$ contain type class constraints. There is an implicit quantifier $\forall \bar{a}$ scopes over $C_1 \land C_2$ 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 $P \models \exists \bar{a}.C_1 \land C_2$ which will supply us with an algorithm to decide $P \models C$ as well.

In [19], we showed how to reduce $P \models \exists \bar{a}.C_1 \land C_2$ to CHR solving. We apply CHRs on $C_1$ and $C_1 \land C_2$ 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 $\text{insert} :: \text{Ord } a \Rightarrow \text{a} \rightarrow \text{[a]} \rightarrow \text{[a]}$. In the translation of insert, the subsumption check boils down to checking $P \models \exists \text{Ord } a \land \text{Advice } N_2 (a \rightarrow [a] \rightarrow [a]) \models (\ast)$

where

$$P = \{ \text{Advice } N_3 \ (\text{Int } \rightarrow [\text{Int } \rightarrow \text{Int}]) \models \text{True},$$

$$\text{Advice } N_2 \text{ a } \models (\text{Int } \rightarrow [\text{Int } \rightarrow \text{Int}]) \models \text{True},$$

$$\text{Ord } \text{Int } \models \text{True} \}$$

We ignore here advice $N_1$ and include the CHR representing the Haskell Prelude instance $\text{Ord } \text{Int}$. The trouble is that none of the CHRs applies to $\text{Advice } N_2 (a \rightarrow [a] \rightarrow [a])$. 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 $(\ast)$ 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 $(\ast)$ by re-solving $\text{Ord } a \models \text{True}$ via the third CHR and $\text{Advice } N_2 (a \rightarrow [a] \rightarrow [a])$ resolves to True via the first CHR. Hence, $(\ast)$ holds for $a = \text{Int}$. In case $a \neq \text{Int}$, $\text{Advice } N_2 (a \rightarrow [a] \rightarrow [a])$ resolves to True via the second CHR. In summary, we have verified $(\ast)$ by case analysis.

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

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

$\text{TC} \; \bar{t} \leftarrow \bar{t} \not\models \bar{t} \mid \text{TC}_1 \; \bar{t}_1, \ldots , \text{TC}_n \; \bar{t}_n$

applies to a constraint $C$ if we find $\text{TC} \; \bar{t} \in C$ such that $\phi(\bar{t}) = \bar{t}$ and $\phi(\bar{t})$ 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

$$C \models C \rightarrow C \land \text{TC} \; \bar{t} \cup \{ \text{TC}_1 \; \phi(\bar{t}_1), \ldots , \text{TC}_n \; \phi(\bar{t}_n) \}$$

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

We write $C \rightarrow C'$ 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
For example, consider

This property is known as coherence [1]. In the type class world, annotations and type classes. We refer [23] for details. There is a large amount of works on the semantics of aspect-oriented programming languages, for example consider [2, 13, 26, 2006/10/3].

We might hope to obtain a completeness result. However, there 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 derivations. The following definitions can be found in similar form in [19].

We say a derivation is unambiguous iff \( \forall \theta : \bigvee t : \Gamma \Rightarrow E \) in the derivation tree are unambiguous.

We say a derivation \( D \) is unambiguous iff all judgments \( C, \Gamma \vdash e : t \Rightarrow E \) in the derivation tree are unambiguous.

We say a derivation \( D_{1} \) with final judgment \( C_{1}, \Gamma \vdash e : t_{1} \) 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 \( \bar{a} = f_{\Sigma}(C_{1}, t_{1}) - f_{\Sigma}(\Gamma) \) and \( \bar{b} = f_{\Sigma}(C_{2}, t_{2}) - f_{\Sigma}(\Gamma) \). In such a situation, we write

\[ C_{1}, \Gamma \vdash e : t_{1} \vdash C_{2}, \Gamma \vdash e : t_{2}. \]

We say 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 \( C_{2}, \Gamma \vdash e : t_{2} \).

We say that a derivation \( D_{1} \) with final judgment \( C_{1}, \Gamma \vdash e : t_{1} \) is principal if 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) \( p : \Gamma \vdash E_{1}, (3) p : \Gamma \vdash E_{2} \) and (4) \( \eta \vdash \Gamma_{\text{init}}. \) 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, 2006/10/3].
We could state concise type soundness, type inference and coherence results. Type class resolution is achieved via CHR solving by use guarded CHRs to represent advice and instead of a dictionary-passing scheme employed in AOP Mini Haskell, we use a type-passing scheme to translate AOP 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 instances 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 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


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].