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
Oxford University Computing Laboratory,
Wolfson Building, Parks Road,
Oxford OX1 3QD, UK
meng.wang@comlab.ox.ac.uk

ABSTRACT

We consider the problem of adding aspects to a strongly typed language which supports type classes. We show that type classes as supported by the Glasgow Haskell Compiler can model an AOP style of programming via a simple syntax-directed transformation scheme where AOP programming idioms are mapped to type classes. The drawback of this approach is that we cannot easily advise functions in programs which carry type annotations. We sketch a more principled approach which is free of such problems by combining ideas from intentional type analysis with advanced overloading resolution strategies. Our results show that type-directed static weaving is closely related to type class resolution – the process of typing and translating type class programs.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Operational semantics*

General Terms

Languages, theory

Keywords

Type class resolution, type-directed weaving

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 [1, 13, 22, 24, 25, 27]. Some researchers have been looking into the connection between AOP and other paradigms such as generic programming [28]. To the best of our knowledge, we are the first to study the connection and combination between AOP and the concept of type classes, a type extension to support overloading (a.k.a. ad-hoc polymorphism) [23, 11], which is one of the most prominent features of Haskell [17].

In this paper, we make the following contributions:

- We introduce an AOP extension of Haskell, referred to as AOP Haskell, with type-directed pointcuts (Section 3.1).
- We define AOP Haskell by means of a syntax-directed translation scheme where AOP programming idioms are directly expressed in terms of type class constructs. Thus, typing and translation of AOP Haskell can be explained in terms of typing and translation of the resulting type class program (Section 5).

Our type class encoding of AOP critically relies on multi-parameter type classes and overlapping instances. Both features are not part of the Haskell 98 standard [17], but they are supported by the Glasgow Haskell Compiler (GHC) [4]. There are two problems.

Firstly, GHC's overlapping instances have never been formalized. Hence, it is difficult to make any precise claims regarding soundness of our GHC type class encoding of AOP. Secondly, the AOP to GHC type class translation scheme only works in case we do *not* advise programs which contain type annotations. Section 4.2 provides further details.

Despite these problems, we consider the encoding of AOP via GHC type classes a useful exercise. To encode AOP in the setting of a strongly typed language we need some form of type-safe cast. Type classes are known to have this capability and in our approach we achieve this by exploiting GHC's overlapping instances. Thus, we can establish that the concepts of type classes and aspects are closely related.

There are a number of works, for example consider [14, 12], which also use sophisticated type class tricks to model type safe casts in the setting of generic programming and strongly typed heterogeneous collections. These works may

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Sixth International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007), March 13, 2007, Vancouver, BC, Canada.
Copyright 2007 ACM ISBN 1-59593-671-4/07/03 ...$5.00.

even have a solution how to advise functions without having to change type annotations by using some more advanced type class "hackery". However, we do not plan to consider this avenue further. Ultimately, we seek for a more principled approach which allows us to study the combination of type classes and aspects without having to rely on the features of specific implementations such as GHC.

We are currently working on a foundational framework to integrate type classes and aspects. We briefly sketch this more principled approach in Section 6. The idea is to use Harper and Morrisett's intentional type analysis framework [6] for the translation of aspects and Stuckey and the first author's overloading framework [19] for the resolution of type classes programs with aspects.

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 Haskell to GHC type classes. Section 4 discusses the shortcomings of this approach. We conclude in Section 7 where we also discuss related work.

2. BACKGROUND: TYPE CLASSES

Type classes [11, 23] provide for a powerful abstraction mechanism to deal with user-definable overloading also known as ad-hoc polymorphism. The basic idea behind type classes is simple. Class declarations allow one to group together related methods (overloaded functions). Instance declarations prove that a type is in the class, by providing appropriate definitions for the methods.

Here are some standard Haskell declarations.

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

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

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

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

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

In the standard type class translation approach we represent each type class via a dictionary [23, 5]. 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. The dictionary translation of the above declarations is given in Figure 1.

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

Figure 1: Dictionary-Passing Translation

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

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

In the above, we only use single-parameter type classes. Other additional type class features include functional dependency [9], constructor [8] and multi–parameter [10] type classes. For the translation of AOP Haskell to Haskell we will use multi-parameter type classes and overlapping instances, yet another type class feature, as supported by GHC [4].

3. THE KEY IDEAS

To explain our idea of how to mimic AOP via GHC type classes, we first introduce an AOP extension of Haskell, referred to as AOP Haskell, and consider some example programs in AOP Haskell.

3.1 AOP Haskell

AOP Haskell extends the Haskell syntax [17] by supporting top-level aspect definitions of the form

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

```
import List(sort)

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

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

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

Figure 2: AOP Haskell Example

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

In Figure 2, we give an example program. In the top part, we provide the implementation of an insertion sort algorithm where elements are sorted in non-decreasing order. At some stage during the implementation, we decide to add some security and optimization aspects to our implementation. We want to ensure that each call to `insert` takes a sorted list as an input argument and returns a sorted list as the result.

In our AOP Haskell extension, we can guarantee this property via the first aspect definition in Figure 2. We make use of the (trusted) library function `sort` which sorts a list of values. The `sort` functions assumes the overloaded comparison operator `<=` which is part of the `Ord` class. Hence, we find the pointcut type `Ord a=>[a]->[a]->[a]`. The keyword `proceed` indicates to continue with the normal evaluation. That is, we continue with the call `insert x ys`. The second aspect definition provides for a more efficient implementation in case we call `insert` on list of `Int`s. We assume that only non-negative numbers are sorted which implies that `0` is the smallest element appearing in a list of `Int`s. Hence, if `0` is the first element it suffices to cons `0` to the input list. Notice there is an overlap among the

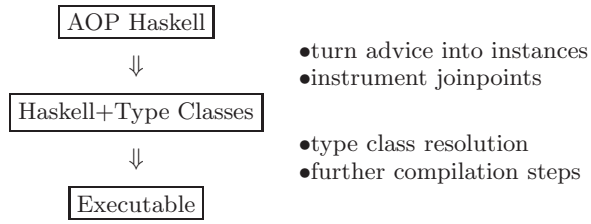

Figure 3: AOP Haskell Typing and Translation Scheme

pointcut types for `insert`. In case we call `insert` on list of `Int`s we apply both advice bodies in no specific order unless otherwise stated. For all other cases, we only apply the first advice.

Because AOP Haskell extends Haskell, we can naturally refer to overloaded functions in advice bodies. See the first advice body where our use of `sort` gives rise to the `Ord a` constraint. Also note the use of the (overloaded) equality operator `==` whose type is `Eq a => a -> a -> a`. In Haskell, the `Eq` class is a superclass of `Ord`. Hence, there is no need to mention the `Eq` class in the pointcut type of the advice definition.

3.2 Typing and Translating AOP Haskell with GHC Type Classes

Our goal is to embed AOP Haskell into Haskell by making use of Haskell's rich type system. Specifically, we use GHC with two extensions (multi-parameter type classes and overlapping instances). We give a transformation scheme where typing and translation of the *source* AOP Haskell program is described by the resulting *target* Haskell program.

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 such as type inference and type soundness.

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.

Figure 3 summarizes our approach of typing and translating AOP Haskell. In Figure 4, we apply the transformation scheme to the AOP Haskell program from Figure 2. 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

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

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

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

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

Figure 4: GHC Haskell Translation of Figure 2

```
joinpoint N1 (joinpoint N2 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].

Next, each piece of advice is turned into an instance declaration where the type parameter `n` of the `Advice` class is set to the singleton type of the advice and type parameter `t` is set to the pointcut type. In case the pointcut type is of the form `C => ...`, we set the instance context to `C`. See the translation of advice `N1`. In the instance body, we simply copy the advice body where we replace `proceed` by the name of the advised function. Additionally, for each advice `N` we introduce `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. For example, the head `Advice N1 (a->[a]->[a])` of instance (I1) and the head `Advice N1 a` of the default instance (I1') overlap because the type components are unifiable (after renaming the `a` in `Advice N1 a` with a fresh variable `b`). Therefore, we can potentially use either of the two instances to resolve a type class constraint which may yield to two different results. However, GHC will postpone resolution of type classes until we can unambiguously choose an instance. We say that GHC implements a "lazy" and "best-fit" type class resolution strategy.

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

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

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

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

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

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

GHC's "best-fit" strategy resolves `Advice N1 (Int -> [Int] -> [Int])` via instance (I1), `Advice N2 (Int -> [Int] -> [Int])` via instance (I2) and `Ord Int` is resolved using a pre-defined instance from the Haskell Prelude [17]. Effectively, this means that at locations (1) and (2) in the above program text, we intercept the calls to `insert` by first applying the body of instance (I1) followed by applying the body of instance (I2)

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

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

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

4. DISCUSSION

The transformation from AOP Haskell to Haskell using GHC type classes is simple and only requires a syntactic transformation of programs. In Section 5, we give the details plus further examples. We also show how to statically detect useless advice. Unfortunately, our AOP to type class transformation scheme suffers from the following problems:

1. Aspects must be pure, i.e. free of side-effects.
2. (a) Advising type annotated requires to rewrite annotations. (b) Rewriting of type annotations of polymorphic recursive functions is impossible.
3. The transformation scheme relies on multi-parameter type classes and overlapping instances extensions which are not part the Haskell 98 standard. But they are supported by GHC.

We will discuss each of the above three issues in turn.

4.1 Aspects Must be Pure

In Haskell the effect of a program is manifested in its (monadic) type. For example, a program which reads and writes to standard I/O will have type `IO ()` where `IO` belongs to the monad class. Hence, based on the syntax-directed transformation scheme described so far, aspects cannot make pure functions do I/O (for example, to do logging) or modify state (for example, to add memorization). We would need to "semantically rewrite" the program during the transformation, by for example changing an advised function of type `t` to a function of type `IO t` in case of an aspect with effect `IO`.

A possible systematic solution is to monadify programs [2] and use (state) monad transformers [15]. Another alternative is to use `unsafePerformIO` which obviously breaks type safety. We consider the issue of impure aspects as orthogonal to our work is which about establishing a connection between AOP and the concept of type classes. We plan to take a look at impure aspects in future work.

4.2 Advising Type Annotated Programs

Let us assume we provide explicit type annotations to the functions in Figure 2.

```
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 behavior. GHC's type class resolution mechanism will "eagerly" resolve the constraints

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

arising from

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

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

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

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

The need for rewriting type annotations complicates our simple AOP Haskell to Haskell transformations. In fact, in case of polymorphic recursive functions, which demand type annotations to guarantee decidable type inference [7], we are unable to appropriately the type annotation.

Let us consider a (contrived) program to explain this point in more detail. In Figure 5, 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. Notice that the lists in the recursive call are getting "deeper' and "deeper". The advice definition allows us to intercept all calls to `f` on list of list of `Bool`s to ensure termination for at least some values.

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

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

Figure 5: Advising Polymorphic Recursive Functions

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

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

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

A closer analysis shows that the problem we face is due to the way type classes are implemented in GHC via the dictionary-passing scheme [5]. In fact, almost all Haskell implementations use the dictionary-passing scheme. Hence, the following observation applies to pretty much all Haskell implementations. In the dictionary-passing scheme, each each type class is represented by a dictionary containing the method definitions. In our case, dictionaries represent the advice which will be applied to a joinpoint. Let us assume we initially call `f` with a list of `Bool`s. Then, the default advice applies and we proceed with `f`'s evaluation. Subsequently, we will call `f` on a list of list of `Bool`s. Recall that `f` is a polymorphic recursive function. Now, we wish that the advice `N` applies to terminate the evaluation with result `False`. The problem becomes now clear. The initial advice (i.e. dictionary) supplied will need to be changed during the evaluation of function `f` We cannot naturally program this behavior via GHC type classes.

4.3 Transformation Requires Type Class Extensions

To encode AOP in the setting of a strongly typed language we need some form of type-safe cast. Multi-parameter type classes are not essential but GHC style overlapping instances are essential. However, GHC style overlapping instances are heavily debated and still lack a formal description.

4.4 Short Summary

The AOP Haskell to Haskell transformation scheme based on GHC type classes is simple. The problem is that we cannot advise programs which contain type annotations, unless we manually rewrite type annotations. This is impossible in case we advice polymorphic recursive functions. The source of the problem is the dictionary-passing scheme which underlies the translation of type classes in GHC.

A less well known fact is that there exist alternative type class translation proposals based on a type-passing translation scheme [21, 6]. The key insight is that if we employ a type-passing scheme for the translation of aspects we

can easily solve the problems of the GHC based translation scheme. We sketch such an approach in Section 6. First, we provide the details of mapping AOP Haskell to Haskell using GHC type classes.

5. AOP GHC HASKELL

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 [17]. 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 4.2, we impose the following condition on the AOP extension of GHC.

Definition 1 (AOP GHC Haskell Restriction). *We demand that inside the lexical scope of a type annotation, advice or instance declaration there are no joinpoints.*

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 GHC Haskell.

5.1 Type Class-Based Transformation Scheme

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

Definition 2 (AOP to GHC Transformation). *Let p be an AOP Haskell program. We perform the following transformation steps on p to obtain the program p'.*

Advice class: *We add the class declaration*

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

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 _ proceed = e
instance Advice N a -- resolves to default case
```

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

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

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

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

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

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

Claim 1. *Type soundness and type inference for AOP GHC Haskell 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 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. The advice is useful if the program text resulting from the transformation step below is well-typed.

Useful Advice: Each AOP Haskell statement

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

generates

```
eq :: a -> a -> a
eq = undefined
f1' :: C => t
f1' = undefined
f1'' = eq f1 f1'
...
fn' :: C => t
fn' = undefined
fn'' = eq fn fn''
```

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

We may be tempted to generate the following simpler program text.

```
fi' :: C => t
fi' = fi
```

This will work for the above program. But such a transformation scheme is too restrictive as the following example shows.

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

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

Figure 6: Advising Accumulator Recursive Functions

```
module CollectsLib where

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

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

Figure 7: Collection Library

```
g :: [a] -> Int

N@advice #g# :: [a] -> a = ...
```

The advice is clearly useful (in case `a` is `Int`. However, the program text

```
g' :: [a] -> a
g' = g
```

is ill-typed because the annotation is too polymorphic.

The idea behind the useful advice transformation step is to test whether the combination of type constraints from `f1` and `C => t` is consistent (i.e. well-typed). Then, the advice must be useful.

5.2 AOP GHC Haskell Examples

We take a look at a few AOP GHC Haskell 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 6. 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 GHC Haskell, we can catch the base case via the advice `N`. It

```
module Main where

import List(sort)
import CollectsLib

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

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

N2@advice ♯insert♯ ::  Int -> [Int] -> [Int] =
  \x -> \ys -> if x == 0 then x:ys
               else proceed x ys
```

Figure 8: Advising Overloaded Functions

is safe here to give append and reverse type annotations, although, the joinpoint is then enclosed by a type annotation. The reason is that only one advice `N` applies here.

Advising overloaded functions. In our next example, we will show that we can even advise overloaded functions. We recast the example from Section 3.1 in terms of a library for collections. See Figures 7 and 8. 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 8 to the code shown in Figure 4. The code in Figure 7 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 8 by the code in Figure 7 (dropping the statement `module CollectsLib where` of course). Then, we face the problem of advising a function enclosed by a "type annotation". Recall that instance declarations act like type annotations and there is now a joinpoint `insert` within the body of the instance declaration in scope. Our automatic transformation scheme in Definition 2 will not work here. The resulting program may type check but we risk that the program will show some "unaspect"-like behavior. The (programmer-guided) solution is to manually rewrite the instance declaration during the transformation process which roughly yields the following result

```
...
instance (Advice N1 (a->[a]->[a]),
          Advice N2 (a->[a]->[a]),
          Ord a) => Collects [a] a where
```

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


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

Figure 9: Advising functions in advice bodies

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

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

Figure 10: GHC Haskell Translation of Figure 9

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

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

- `-fallow-undecidable-instances`

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

Advising functions in advice bodies. Given that we translate advice into instances, it should be clear that we can also advise functions in advice bodies if we are willing to "guide" the translation scheme. In Figure 9, we give such an example and its (manual) translation is given in Figure 10. We rely again on the "undecidable" instance extension in GHC.

The last example makes us clearly wish for a system where we do not have to perform any manual rewriting. Of course, we could automate the rewriting of annotations by integrating the translation scheme in Definition 2 with the GHC type inferencer. However, the problem remains that we are unable to advise polymorphic recursive functions. Recall the discussion in Section 4.2.

6. TOWARDS A FRAMEWORK FOR TYPE CLASSES AND ASPECTS

We are currently working on a core calculus to study type classes and aspects. The two key ingredients are (1) a type-directed translation scheme from a calculus with type classes and aspects to a variant of Harper and Morrisett's λ_i^{ML} calculus, and (2) a type inference scheme for type class and aspect resolution based on Stuckey and the first author's overloading framework.

We illustrate the key ideas behind this approach via a simple example. We consider parts of the earlier program in Figure 2.

```
import List(sort)
insert ::  Ord a => a -> [a] -> [a]
insert x [] = []
insert x (y:ys) =
  if x <=y then x:y:ys else y :  insert x ys
N1@advice #insert# ::  Ord a => a -> [a] -> [a] = ...
N2@advice #insert# ::  Int -> [Int] -> [Int] = ...
```

We leave out the `insertionSort` function and also omit the advice bodies for brevity. Notice that `insert` carries a type annotation. Earlier we saw that in AOP GHC Haskell we cannot easily advise type annotated functions unless we rewrite type annotations.

We can entirely avoid rewriting of type annotations by switching to a type-passing translation scheme for the translation of advise. Type classes can be translated using the standard dictionary-passing scheme. Here is the translation of the above program.

```
insert = Λ a.  λ d:DictOrd a.  λ x:a.  λ xs:[a].
  case xs of
    [] → [x]
    (y:ys) →
       if (d (<=)) x y then x:y:ys -- (1)
       else y :  (
            (joinpoint N1 (a->[a]->[a]) d -- (2)
            ((joinpoint N2 (a->[a]->[a])) (insert a d)))
            x ys)

joinpoint = Λ n.  Λ a.
  typecase (n,a) of
    (N1,a->[a]->[a]) → λ d:DictOrd a.  ...  -- (3)
    (N1,_) → ...
    (N2,Int->[Int]->[Int]) → ...
    (N2,_) → ...
```

In the translation, function `insert` expects an additional type and dictionary argument. Dictionaries serve the usual purpose. For example, at location (1) the program text `d (<=)` selects the greater-or-equal method from the dictionary of the `Ord` class. The novelty are the additional type arguments which we use for selecting the appropriate advice. See locations (2) and (3).

Similarly to AOP GHC Haskell, we instrument joinpoints with calls to the weaving function `joinpoint`. The difference to AOP GHC Haskell is that we rely on run-time type information and use type case, which is part of the λ_i^{ML} calculus, to select the advice. For example, the advice `N1` is selected using the first two (type) arguments. The (translated) advice body assumes a third (dictionary) argument, see location (2), because we make use of the `Ord` class in

the advice body. Recall the full definition of advice `N1` in Figure 2. At the call site of the weaving function, we must of course supply the necessary arguments. See location (1).

The translation of programs is driving by type inference. To obtain a type inference algorithm, we map type class and advice declarations to Constraint Handling Rules (CHRs) [3]. In [19], Stuckey and the first author introduced an overloading framework where CHRs are used to reason about type class relations. By mapping type classes to CHRs, we can concisely reason about their type inference properties. CHRs have a simple operational semantics and thus we obtain a type inference algorithm.

For example, the type class instance

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

is mapped to the CHR

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

Logically, the symbol `<==>` stands for bi-implication. Operationally, the symbol `<==>` indicates a rewrite relation among constraints (from left to right). In contrast to Prolog, we only perform matching but *not* unification when rewriting constraints. Rewriting of constraints with respect to instances is also known as "context reduction" in the type class literature.

Here, we employ CHRs to reason about advice declarations. The advice declarations of our running example translate to the CHRs

```
Advice N1 (a->[a]->[a]) <==> Ord a
Advice N1 b <==> b /=(a->[a]->[a]) | True
Advice N2 (Int->[Int]->[Int]) <==> True
Advice N2 b <==> b /= (Int->[Int]->[Int]) | True
```

The first and third CHR result from the advice declarations `N1` and `N2` whereas the second and fourth CHR cover the "default" cases. Notice that the second and fourth CHR contain guard constraints which impose additional conditions under which a CHR can fire. For example, the second CHR will only rewrite `Advice N1 b` to `True` (the always true constraint), if `b` is not an instance of `(a->[a]->[a])`. The idea is that via guard constraints we can guarantee that the CHR representing the advice declaration and the default case do not overlap.

We will use the CHRs resulting from type class and advice declarations to solve (i.e. rewrite) constraints arising during type inference. In the translation, `y : insert x ys` is translated to

```
y : ((joinpoint N1 (a->[a]->[a]) d
((joinpoint N2 (a->[a]->[a])) (insert a d)))
x ys)
```

and this program text gives rise to

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

The first constraint arise from the call to `insert`. The second and third constraint arise from the instrumentation. The surrounding program text carries the type annotation `Ord a => a->[a]->[a]`. To check that the type annotation is correct we must perform a *subsumption* test among types which boils down to a *entailment* test among constraints. For our example, we must verify that `Ord a, Advice N1 (a->[a]->[a]), Advice N2 (a->[a]->[a])` follow from `Ord a`. In general, we must verify that the constraints C_1 from the annotation *entail* the constraints C_2 arising from the program text of that annotation, written $C_1 \supset C_2$.

The entailment check is fairly standard for type classes. We exhaustively rewrite C_2 with respect to the instances (i.e. CHRs) until we reach the from C_1, written $C_2 \rightarrowtail^* C_1$. In the presence of advice, this entailment checking strategy will not work anymore because none of the above CHRs applies to `Advice N2 (a->[a]->[a])`. But if we interpret the above CHRs as some logical formula P we find that `Advice N2 (a->[a]->[a])` is a logical consequence of any first-order model of P and `Ord a`. We can verify this fact via a simple case analysis. If `a` equals to `Int` then `Advice N2 (a->[a]->[a])` is equivalent to `True` because of the third CHR. Otherwise, via the fourth CHR we can conclude that `Advice N2 (a->[a]->[a])` is yet again equivalent to `True`. Hence, `Advice N2 (a->[a]->[a])` is a logical consequence of P.

Our idea is to extend the standard rewriting-based approach to check entailment by incorporating a case analysis. Of course, we need to guarantee that the extended entailment check remains decidable for Haskell 98 type classes and advice declarations making use of such type classes.

7. CONCLUSION AND RELATED WORK

There is a large amount of works on the semantics of aspect-oriented programming languages, for example consider [1, 13, 22, 24, 25, 27] and the references therein. There have been only a few works [24, 1, 16] which aim to integrate AOP into ML style languages. We yet have to work out the exact connections to these works. For instance, the work described in [1] 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 [27, 26] gives a a static weaving scheme for a strongly typed functional AOP language via a type-directed translation process. In [27] (Section 6), the authors acknowledge that their type-directed translation scheme for advice is inspired by the dictionary-passing translation for type classes. But they believe that aspects and type classes substantially differ when it comes to typing and translation. In this paper, we confirm that there is a fairly tight connection between aspects and type classes. The system described in [27, 26] does not assume type annotations and therefore we can express all examples from [27, 26] in terms AOP GHC Haskell (Section 5).

The main result of our work is that static weaving for strongly typed languages bears a strong resembles to 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 5). We use GHC style overlapping instances to encode a form of type-safe which is is used to advice functions based on their types. The approach has a number of problems. For example, we cannot easily advise functions in programs with type annotations.

We are in the process of formalizing a more principled approach to integrate type classes and aspects (Section 6). We expect to report results in the near future. Further future work includes the study of effect-full advice which we can represent via monads in Haskell. We also want to consider more complex pointcuts.

Acknowledgments

We thank Andrew Black, Ralf Lämmel, and referees for AOSD'07 and FOAL'07 for their helpful comments on previous versions of this paper.

8. REFERENCES

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

[2] M. Erwig and D. Ren. Monadification of functional programs. *Sci. Comput. Program.*, 52(1-3):101–129, 2004.

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

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

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

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

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

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

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

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

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

[12] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

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

[14] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *Proc. of ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, pages 26–37. ACM Press, 2003.

[15] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proc. of POPL '95*, pages 333–343. ACM Press, 1995.

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

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

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

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

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

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

[22] D. B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *Proc. of AOSD'03*, pages 158–167. ACM Press, 2003.

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

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

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

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

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

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