Aspect-Oriented Programming with Type Classes

Martin Sulzmann National University of Singapore

Meng Wang *Oxford University*

What's this talk about?

- Aspect-oriented programming (AOP) is an emerging paradigm to aid the user in the modularization of cross-cutting concerns.
- Type classes are an established concept to support ad-hoc polymorphism.
- Both concepts have been so far studied in isolation.
- We will see that type classes support AOP to some extent.
- Main observation:

Translation of type classes
$$\approx$$
Type-directed static weaving

type classes $\approx$ *C++ templates* $\approx$ *Java interfaces*

Outline

- AOP
- Type classes in Haskell
- AOP via type classes
- Limitations
- Conclusion and future work

Our running example

- We define a sorting library using the insertion sort algorithm.
  - We need an `insert` function which inserts an element into a sorted list.
  - Easy to program using object-oriented, functional languages.
- At some later stage we want to extend the library via some efficiency and security “aspects”. For this we need AOP.

Object-oriented solution

```
public static <T>
void insertionSortGeneric(T[] a, Comparator<? super T> c)
  for (int i=1; i < a.length; i++) {
      /* Insert a[i] into the sorted sublist */
    T v = a[i];
    int j;
    for (j = i - 1; j >= 0; j--) {
      if (c.compare(a[j], v) <= 0) break;
      a[j + 1] = a[j];
    }
    a[j + 1] = v;
 }
}
```

Functional solution

In Haskell, we can implement insertion sort as follows.

```
module Sorting where

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

insertionSort _ [] = []
insertionSort leq xs =
  insert leq (head xs) (insertionSort leq (tail xs))
```

`insert` takes as an additional argument a function `leq` to check for “lesser than or equal”.

Clumsy, we have to thread through `leq`*.*

Type classes

Excerpts of the Haskell Prelude.

```
module Prelude where
class Eq a where
 (==) :: a -> a -> Bool
class Eq a => Ord a where
 (<=) :: a -> a -> Bool
instance Eq Int where ...
instance Eq a => Eq [a] where ...
instance Ord Int where ...
instance Ord a => Ord [a] where ...
```

- `(==)` is an overloaded method belonging to the type class Eq.
- `Eq t` states that the type `t` is a member of Eq.
- We declare membership via instances.
- We can extend the class hierarchy by introducing subclasses.

Type class solution

```
module Sorting where
import Prelude

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))
```

- Compare the difference to the functional solution. Instead of `leq` we find <= (implicit argument).
- Indeed, type inference yields

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

The challenge

At some stage during the implementation, we decide to add some security and optimization aspects to our implementation.

- Efficiency aspect:
  We know that only non-negative numbers are ever sorted. Hence, if we insert `0` it suffices to cons `0` to the input list.
- Security aspect:
  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.

How to do this (without “affecting” the entire program).

We only want to advise the relevant program parts.

AOP Haskell example

```
-- 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
```

The new keyword `proceed` indicates continuation of the normal evaluation process.

AOP Haskell

- Extension of Haskell with aspect definitions of the form

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

- `N` is the name of the aspect. `f1`,...,`fn` refer to function symbols (the *pointcut*). Each `fi` is referred to as a *joinpoint*.
- Each pointcut has a type annotation `C => t` which follows the Haskell syntax for types.
- The advice body e follows the Haskell syntax for expressions.
- We will apply the (around) advice if the type of a joinpoint `fi` is an instance of `t` such that constraints `C` are satisfied (pointcuts are type directed).

We will see later how to encode AOP Haskell in Haskell.

AOP Haskell

- Advice declarations may refer to overloaded methods and we may advise overloaded methods.
- Aspects must be pure.
- Simple pointcut model.
- Type-directed static weaving.

Sample evaluation (a.k.a. weaving)

Suppose we encounter the function call

```
insert 'b' ['a','c']
```

We use `insert` at type instance `Char->[Char]->[Char]`.

The sortedness aspect applies (pointcuts are type-directed!).

```
-- 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] = ...
```

Sample evaluation (a.k.a. weaving)

Suppose we encounter the function call

```
insert 'b' ['a','c']
```

Hence,

```
    insert 'b' ['a','c']

--> let zs = insert 'b' ['a','c']
    in if (isSorted ['a','c']) && (isSorted zs)
       then zs else error "Bug"

-->* ['a','b','c']
```

How to type and translate AOP Haskell

Our idea: We translate AOP idioms to type classes as supported by the Glasgow Haskell Compiler (GHC).

Specifically,

- Turn advice into instances.
- Instrument joinpoints with calls to a “weaving” function.

Turning advice into instances

```
class Advice n a where
   joinpoint :: n -> a -> a
   joinpoint _ = \x -> x      -- default instance
data N1 = N1
data N2 = N2
-- N1@advice #insert# :: Ord a => a -> [a] -> [a] = ...
instance Ord a => Advice N1 (a->[a]->[a]) where ...
instance Advice N1 a
-- N2@advice #insert# :: Int -> [Int] -> [Int] = ...
instance Advice N2 (Int->[Int]->[Int]) where ...
instance Advice N2 a
```

- `joinpoint` is the (overloaded) weaving function.
- `N1` and `N2` are singleton types.

We will shortly discuss the overlap among the instances

Turning advice into instances

In detail,

```
-- 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
```

is turned into

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

Instrumenting joinpoints

Each call to `insert` is replaced by

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

We assume here the following order among advice: $\mathtt{N2} \leq \mathtt{N1}$.

If `insert` is used at the type instance `a->[a]->[a]`, then the above gives rise to

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

Hence, after instrumentation function `insert` has type

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

We need to take a look at type class resolution now.

Type class resolution

In case of

```
instance Eq a => Eq [a] where ...
instance Eq Int where ...
```

`Eq [Int]` resolves to `Eq Int` via the first instance and then subsequently to `True` via the second instance.

`Eq [Int]` refers to a use of `(==)` at the type instance `[Int]->[Int]->Bool`.

Hence, type class resolution tells us how to build the concrete instance of `(==)` requested at the type `[Int]->[Int]->Bool`.

Type class resolution

In detail,

```
instance Eq a => Eq [a] where ...
instance Eq Int where ...
```

translates to

```
data DictEq a = (a->a->Bool)
instI1 :: DictEq Int
instI2 :: DictEq a -> DictEq [a]
```

The dictionary `instI2 instI1` provides evidence for `Eq [Int]`.

Overlapping instances resolution

In case of

```
instance Ord a => Advice N1 (a->[a]->[a]) -- (A1)
instance Advice N1 a
instance Advice N2 (Int->[Int]->[Int])    -- (A2)
instance Advice N2 a
```

and

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

we cannot deterministically resolve the above type class constraints. Hence, we leave them unresolved.

However, in case a=`Int` we can apply the “best-fit” strategy strategy. `Advice N2 (Int->[Int]->[Int])` is resolved via instance `(A1)`.
`Advice N1 (Int->[Int]->[Int])` resolves to `Ord Int` which then resolves to `True`.

Type classes vs type-directed weaving

Assume we use (the instrumented program)

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

at type instance `Int->[Int]->[Int]`.

Type class resolution will then replace the calls to the "weaving" function `joinpoint` with calls to the appropriate advice bodies.

We assume here type classes as supported by the Glasgow Haskell Compiler (GHC).

We conclude

$$\text{Translation of type classes} \\ \approx \\ \text{Type-directed static weaving}$$

Limitations

Assume the instrumented program carries a type annotation.

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

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

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

which arise from location (1) via

```
instance Ord a => Advice N1 (a->[a]->[a]) --(A1)
instance Advice N2 a
```

Unexpected behavior.

Limitations

We need to manually change type annotations.

Replace

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

by

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

Clumsy and even impossible in case of polymorphic recursive functions, see paper for details.

But the approach works for Hindley/Milner + type classes.

Related work

- Work on the semantics of AOP:
  Chen, Dantas, Dutchyn, Khoo, Kiczales, Krishnamurthi, Lämmel, Ligatti, Tucker,Walker, Wand, Wang, Washburn, Weirich, Zdancewic
  [DWWW05, Läm02, TK03, WZL03, WKD04, WCK06b]
- Work on AOP in the context of ML style languages:
  Dantas, Ligatti, Masuhara,Tatsuzawa, Walker, Washburn,Weirich,
  Yonezawa,Zdancewic [WZL03, DWWW05, MTY05]
- Work on type class encoding tricks:
  Kiselyov, Lämmel,
  Peyton Jones,Schupke [LP03, KLS04]

Conclusion

- AOP GHC Haskell: A light-weight form of AOP with GHC style overlapping instances.
- Syntax-directed translation scheme from AOP GHC Haskell to GHC Haskell.
- Limitation: We cannot advise programs which contain type annotations (but the approach works for Hindley/Milner + type classes).
- AOP GHC Haskell can deal with all examples from [WCK06b, WCK06a].
- Observation: Type-directed static weaving is closely related to type class resolution – the process of typing and translating type class programs.

Future work

Towards a framework for type classes and aspects.

Key observations:

- Type classes are open.

  Type class resolution via forward chaining

- Aspects are closed.

  Type class resolution via backward chaining/search

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.

A more principled approach

```
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] =
```

- `insert` carries now a type annotation (to translate using overlapping type classes we need to manually rewrite type annotations).
- First key idea: We use the standard dictionary-passing translation scheme for type classes but use a type-passing scheme for aspects.

A more principled approach

The translation yields

```
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,_) → ...
```

A more principled approach

Second key idea: Type class resolution may now involve a search (because aspects are closed).

We employ Constraint Handling Rules (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
```

`insert`'s annotation provides `Ord a` and the (instrumented) program text demands

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

Perform case analysis, i.e. solving by search.

References

[DWWW05] 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.

[KLS04] 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.

[Läm02] 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.

[LP03] 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.

[MTY05] 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.

[TK03] D. B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *Proc. of AOSD'03*, pages 158–167. ACM Press, 2003.

[WCK06a] M. Wang, K. Chen, and S.C. Khoo. On the pursuit of staticness and coherence. In *FOAL '06: Foundations of Aspect-Oriented Languages*, 2006.

[WCK06b] 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.

[WKD04] 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.

[WZL03] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *Proc. of ICFP'03*, pages 127–139. ACM Press, 2003.