ML Modules and Haskell Type Classes: A Constructive Comparison

Stefan Wehr[1]   Manuel M. T. Chakravarty[2]

[1] University of Freiburg, Germany   [2] University of New South Wales, Australia

APLAS; December 10, 2008; Bangalore, India

Motivation

- Formal comparison of ML modules with Haskell type classes
- Support for overloading:
  - excellent in Haskell
  - rudimentary in ML
- Module system:
  - weak in Haskell
  - powerful in ML

Motivation

- Formal comparison of ML modules with Haskell type classes
- Support for overloading:
  - excellent in Haskell
  - rudimentary in ML
- Module system:
  - weak in Haskell
  - powerful in ML
- see also `http://augustss.blogspot.com/2008/12/somewhat-failed-adventure-in-haskell.html`

Translating ML modules to Haskell Type Classes

Translating signatures and structures

```
structure IntSet =
struct
    type elem       = int
    type set        = elem list
    val  empty      = []
    fun  member i s = any (intEq i) s
    fun  insert i s = if member i s then s else (i::s)
end
```

Translating signatures and structures

```
structure IntSet =
struct
    type elem       = int
    type set        = elem list
    val  empty      = []
    fun  member i s = any (intEq i) s
    fun  insert i s = if member i s then s else (i::s)
end
```

```
class SetSig a where
    type Elem   a
    type Set    a
    empty  ::   a -> Set a
    member ::   a -> Elem a -> Set a -> Bool
    insert ::   a -> Elem a -> Set a -> Set a

data IntSet = IntSet

instance SetSig IntSet where
    type Elem IntSet = Int
    type Set  IntSet = [Int]
    empty _          = []
    member _ i s     = any (intEq i) s
    insert _ i s     = if member IntSet i s then s else (i : s)
```

Translating signatures and structures

```
structure IntSet =
struct
    type elem       = int
    type set        = elem list
    val  empty      = []
    fun  member i s = any (intEq i) s
    fun  insert i s = if member i s then s else (i::s)
end
```

```
class SetSig a where
    type Elem   a
    type Set    a
    empty  ::  a -> Set a
    member ::  a -> Elem a -> Set a -> Bool
    insert ::  a -> Elem a -> Set a -> Set a
```

associated type synonym

```
data IntSet = IntSet

instance SetSig IntSet where
    type Elem IntSet = Int
    type Set  IntSet = [Int]
    empty  _         = []
    member _ i s     = any (intEq i) s
    insert _ i s     = if member IntSet i s then s else (i : s)
```

Translating abstract types

```
structure IntSet' = IntSet :>
sig
    type elem   = int
    type set
    val  empty  : set
    val  member : elem -> set -> bool
    val  insert : elem -> set -> set
end
```

Translating abstract types

```
structure IntSet' = IntSet :>
sig
    type elem   = int
    type set
    val  empty  : set
    val  member : elem -> set -> bool
    val  insert : elem -> set -> set
end
```

```
data IntSet' = IntSet'

instance SetSig IntSet' where
    type    Elem IntSet' = Elem IntSet
    abstype Set  IntSet' = Set IntSet
    empty  _             = empty IntSet
    member _             = member IntSet
    insert _             = insert IntSet
```

Translating abstract types

```
structure IntSet' = IntSet :>
sig
    type elem   = int
    type set
    val  empty  : set
    val  member : elem -> set -> bool
    val  insert : elem -> set -> set
end
```

```
data IntSet' = IntSet'

instance SetSig IntSet' where
    type     Elem IntSet' = Elem IntSet
    abstype Set  IntSet' = Set IntSet
    empty  _             = empty IntSet
    member _             = member IntSet
    insert _             = insert IntSet
```

abstract associated type synonym

Translating functors

```
functor MkSet (E : sig
                    type t
                    val eq : t -> t -> bool
                  end) =
struct
  type elem = E.t
  type set = E.t list
  val empty = []
  fun member x s = any (E.eq x) s
  fun insert x s = if member x s then s else (x :: s)
end
```

Translating functors

```
functor MkSet (E : sig
                    type t
                    val eq : t -> t -> bool
                  end) =
struct
  type elem = E.t
  type set = E.t list
  val empty = []
  fun member x s = any (E.eq x) s
  fun insert x s = if member x s then s else (x :: s)
end
```

```
class EqSig a where
    type T a
    eq :: a -> T a -> T a -> Bool
class EqSig a => MkSetSig b a where
    type Elem' b a
    type Set' b a
    empty' :: b -> a -> Set' b a
    member' :: b -> a -> T a -> Set' b a -> Bool
    insert' :: b -> a -> T a -> Set' b a -> Set' b a
data MkSet = MkSet
instance EqSig a => MkSetSig MkSet a where
    type Elem' MkSet a = T a
    type Set'  MkSet a = [T a]
    empty' _ _     = []
    member' _ a x s = any (eq a x) s
    insert' _ a x s = if member' MkSet a x s then s else (x : s)
```

Translating functor applications

```
structure StringSet = MkSet(struct
                              type t  = string
                              val  eq = stringEq
                            end)
```

Translating functor applications

```
structure StringSet = MkSet(struct
                              type t  = string
                              val  eq = stringEq
                            end)
```

```
data StringEq = StringEq

instance EqSig StringEq where
    type T StringEq = String
    eq _ = stringEq

data StringSet = StringSet

instance SetSig StringSet where
    type Elem StringSet = Elem’ MkSet StringEq
    type Set  StringSet = Set’ MkSet StringEq
    empty _  = empty’ MkSet StringEq
    member _ = member’ MkSet StringEq
    insert _ = insert’ MkSet StringEq
```

Summary of translation ML ⟶ Haskell

ML	**Haskell**
structure signature	one-parameter type class
structure	instance of the corresponding type class
name of structure/functor	data type
functor argument signature	single-parameter type class
functor result signature	two-parameter type class
functor	instance of the result class
type component	associated type synonym
value component	method

Differences

- Namespace management
  ML: yes — Haskell: no
- Signature/structure components
  ML: all sorts of language constructs — Haskell: methods only (with extensions: type components)
- Implicit vs. explicit signatures
  ML: implicit — Haskell: explicit
- Anonymous vs. named signatures/structures
  ML: anonymous — Haskell: named
- Signature matching
  ML: structural — Haskell: nominal
- First-class structures
  ML: no (supported by extensions) — Haskell: yes

Translating Haskell Type Classes to ML modules

Translating type class declarations

```
class Eq a where
    eq :: a -> a -> Bool

class Eq a => Ord a where
    lt :: a -> a -> Bool
```

Translating type class declarations

```
class Eq a where
    eq :: a -> a -> Bool

class Eq a => Ord a where
    lt :: a -> a -> Bool
```

```
signature Eq =
sig
  type t
  val  eq : t -> t -> bool
end

signature Ord =
sig
  type t
  val  lt      : t -> t -> bool
  val  superEq : [Eq where type t = t]
end
```

Translating type class declarations

```
class Eq a where
    eq :: a -> a -> Bool

class Eq a => Ord a where
    lt :: a -> a -> Bool
```

```
signature Eq =
sig
  type t
  val  eq : t -> t -> bool
end

signature Ord =
sig
  type t
  val  lt      : t -> t -> bool
  val  superEq : [Eq where type t = t]
end
```

type of a first-class structure

Translating overloaded functions

```
elem :: Eq a => a -> [a] -> Bool
elem x l = any (eq x) l
```

Translating overloaded functions

```
elem :: Eq a => a -> [a] -> Bool
elem x l = any (eq x) l
```

```
fun elem d (x:'a) l =
    let structure D as Eq where type t = 'a
          = d
    in any (D.eq x) l
    end
```

Translating overloaded functions

```
elem :: Eq a => a -> [a] -> Bool
elem x l = any (eq x) l
```

type: `[{type t = 'a, val eq :  'a -> 'a -> bool}]`

```
fun elem d (x:'a) l =
    let structure D as Eq where type t = 'a
          = d
    in any (D.eq x) l
    end
```

unpacks a first-class structure

Translating instance declarations (1/2)

```
instance Eq Int where
    eq = intEq

instance Ord Int where
    lt = intLt
```

Translating instance declarations (1/2)

```
instance Eq Int where
    eq = intEq

instance Ord Int where
    lt = intLt
```

```
functor EqInt()  =
struct
  type t  = int
  val  eq = intEq
end

functor OrdInt() =
struct
  type t       = int
  val  lt      = intLt
  val  superEq = [structure EqInt() as Eq where type t = t]
end
```

Translating instance declarations (1/2)

```
instance Eq Int where
    eq = intEq

instance Ord Int where
    lt = intLt
```

```
functor EqInt()  =
struct
  type t  = int
  val  eq = intEq
end

functor OrdInt() =
struct
  type t       = int
  val  lt      = intLt
  val  superEq = [structure EqInt() as Eq where type t = t]
end
```

packs a structure as a first-class structure

Translating instance declarations (2/2)

```
instance Eq a => Eq [a] where
    eq []     []     = True
    eq (x:xs) (y:ys) = eq x y && eq xs ys
    eq _      _      = False
```

Translating instance declarations (2/2)

```
instance Eq a => Eq [a] where
    eq []     []     = True
    eq (x:xs) (y:ys) = eq x y && eq xs ys
    eq _      _      = False
```

```
structure R =
rec (R': sig
           functor F: functor(X: Eq) -> Eq where type t = X.t list
         end)
  struct
    functor F(X: Eq) = struct
      type t = X.t list
      fun eq []      []      = true
        | eq (x::xs) (y::ys) =
            let structure Y = R'.F(X)
            in X.eq x y andalso Y.eq xs ys
            end
        | eq _       _       = false
    end
  end
functor EqList(X: Eq) = R.F(X)
```

Translating instance declarations (2/2)

```
instance Eq a => Eq [a] where
    eq []     []     = True
    eq (x:xs) (y:ys) = eq x y && eq xs ys
    eq _      _      = False
```

```
structure R =
rec (R': sig
           functor F: functor(X: Eq) -> Eq where type t = X.t list
         end)
  struct
    functor F(X: Eq) = struct
      type t = X.t list
      fun eq []      []      = true
        | eq (x::xs) (y::ys) =
            let structure Y = R'.F(X)
            in X.eq x y andalso Y.eq xs ys
            end
        | eq _       _       = false
    end
  end
functor EqList(X: Eq) = R.F(X)
```

forward declaration of recursive structure

Summary of translation Haskell → ML

Haskell	**ML**
type class declaration	signature
type class method	value specification
superclass	value component storing a dictionary
dictionary	first-class structure
(recursive) instance declaration	(recursive) functor
instance constraint	functor parameter
function constraint	dictionary parameter

Differences

- Overloading resolution
  Haskell: implicit ML: explicit
- Recursive vs. sequential definitions
  Haskell: recursive ML: sequential
- Default implementations in type classes/signatures
  Haskell: yes ML: no

Summary

- Translations from ML modules to Haskell type classes and vice versa
  - Both translations formalized
  - Both translations preserve types
  - Both translations implemented, output runnable under
    - GHC 6.10.1 (change `abstype` to `type`)
    - Moscow ML 2.0.1
- Detailed comparison between ML modules and Haskell type classes