First-Class Modules for Haskell

M. Shields and S. Peyton Jones

Presented by Assaf Kfoury
February 4, 2002

Overview

Bridging the “final gap” between the core language and the module language in Haskell, by adding the following features:

- Record types, with fields of polymorphic type, dot notation, and the ability to use a single field name in distinct record types.
- Nested type declarations inside such records.
- First-class universal and existential quantification. Combined with record types, they allow to conveniently express the types of functors.
- A declaration-oriented construct for opening an existentially-quantified value, together with a notation to allow opened types to appear in type annotations, This is in contrast to the standard approach which is expression-oriented and unbearably clumsy

1. Parameterized Records

Declaring a record type, here parameterized:

```
record Set a f = {
  empty  :: f a
  add    :: a -> f a -> f a
  union  :: f a -> f a -> f a
  asList :: f a -> [a]
}
```

- `f` has kind `Type -> Type`
- equality between record types is nominal (not structural)
- single field name can be re-used in different record types

Constructing a record term, by applying a record constructor to a sequence of (possibly mutually recursive – not here) bindings:

```
intListSet :: Set Int []  {- inferred -}
intListSet = Set {
  empty  = []
  add    = \(x :: Int) xs -> x : filter (/= x) xs
  union  = foldr add
  asList = id
}
```

- polymorphic type of `foldr` is
  `foldr :: (a -> b -> b) -> b -> [a] -> b`
- type signature may be omitted wherever comment "`{- inferred -}`" is inserted.

Record term used in a pattern, with “dot notation” used for field projection:

```
one :: [Int]  {- inferred -}
one = intListSet.asList
        (intListSet.add 1 intListSet.empty)
```

Viewing a module as a record,
with an ML functor now turned into an ordinary function:

```
record EqR a = { eq :: a -> a -> Bool }

mkListSet :: forall a . EqR a -> Set a []
mkListSet eq = Set {
  empty = []
  add = \x xs ->
    x : filter (\y -> not (eq.eq x y)) xs
  asList = id
}
```

Outermost quantifier "`forall a`" optional ??

Mixing “functors” with Haskell’s type class mechanism:

```
mkListSet' :: forall a . Eq a => Set a []
mkListSet' = Set {
   empty = []
   add = \x xs -> x : filter ((/=) x) xs
   asList = id
}
```

Overloaded operator `(/=)` is used to replace explicit parameterization over the record “`EqR a`” with implicit parameterization over the class “`Eq a`”

Record fields that have polymorphic types:

```
record Monad f = {
  fmap :: forall a b . (a -> b) -> f a -> f b
  unit :: forall a . a -> f a
  bind :: forall a b . f a -> (a -> f b) -> f b
}
```

which …

(outermost quantifiers "`forall` …" may be omitted ??)

… can be constructed and taken apart as before:

```
listMonad :: Monad []    {- inferred -}
listMonad = Monad {
   fmap = map
   unit = \a -> [a]
   bind = \ma f -> concat (map f ma)
}

singleton :: a -> [a]
singleton x = listMonad.unit x
```

- No subtyping allowed.
- No extensibility for records allowed.

2. Type Inference

Type inference is problematic. Consider:

```
g = \m f x -> m.fmap f (m.unit x)
```

"`fmap`" may be a field name in many records and the type of "`m.fmap`" depends on the type of "`m`". The latter may not be known when trying to infer a type for "`g`".

Difficulties avoided by imposing the *binder rule*:

The programmer must supply a type annotation for every lambda-bound and every letrec-bound variable whose type mentions a record type constructor.

Binder rule may not be easy to satisfy ….

3. Nested Type Declarations

Record declarations may contain nested type declarations:

```
record BTSet a = {
  data BinTree = Leaf | Node BinTree a BinTree

  empty :: BinTree
  add :: a -> BinTree -> BinTree
}
```

Projecting a nested type from a type using ^ (instead of the usual .) to denote type projection:

```
unitSet :: BTSet a -> a -> (Set a)^BinTree
unitSet set a = set.add a set.empty
```

Another way of writing the signature for add in the above record declaration:

```
add :: a -> (Set a)^BinTree -> (Set a)^BinTree
```

Nested record declarations:

```
record Graph ver = {
  record Edge = { from :: ver; to :: ver }
  data   Rep  = Rep [ver] [Edge]

  mkGraph      :: [ver] -> [edge] -> Rep
  transClosure :: Rep -> Rep
}
```

Referencing nested data and record constructors within terms, using projection syntax:

```
leaf :: forall a . (Set a)^BinTree {- inferred -}
leaf = Set^Leaf
edge :: (Graph Int)^Edge                {- inferred -}
edge = Graph^Edge { from = 1; to = 2 }
```

Record terms whose types contain nested types:

```
trivGraph :: Graph () {- inferred -}
trivGraph = Graph {
    mkGraph = \vs es ->
      Rep [()] [Edge { from = (); to = () }]
    transClosure = \r -> r
  }
```

Two critical restrictions:

1. Type declarations are not allowed in record terms, thus avoiding the need for dependent types.

2. Abstract types (i.e., types which are named but whose definition is hidden) are not allowed in record types.

Nested type declarations can always be flattened into non-nested declarations:

```
data BTSet_BinTree a
  = BTSet_Leaf
  | BTSet_Node (BTSet_BinTree a) a
               (BTSet_BinTree a)

record BTSet a = {
  empty :: BTSet_BinTree a
  cmp :: a -> a -> BTSet_Cmp
  add :: a -> BTSet_BinTree a -> BTSet_BinTree a
```

4. Existentials

Hiding the implementation of sets as lists, we can use existential quantifiers:

```
intSet :: exists f . Set Int f
intSet = Set {
  empty  = []
  add    = \x xs -> x : filter ((/=) x) xs
  asList = id
}
```

Existential quantifiers can occur in the result of a function type:

```
mkListSet :: forall a .
                EqR a -> exists f . Set a f

mkListSet' :: forall a . Eq a =>
                  exists f . Set a f
```

Type inference for rank-2 forall-polymorphism,
Already implemented for Haskell. For the term:

```
(\(f :: forall a . a -> Int -> a) -> f 1 2)
  (\x y -> x)
```

the system finds that `(\x y -> x)` has most general type:

```
forall b c . b -> c -> b.
```

This implies the subsumption:

$$\texttt{forall b c . b -> c -> b} \quad \leq \quad \texttt{forall a . a -> Int -> a}$$

The check proceeds by skolemizing the RHS:

$$\texttt{forall b c . b -> c -> b} \quad \leq \quad \texttt{a' -> Int -> a'}$$

then freshening the LHS quantifier variables:

$$\texttt{b' -> c' -> b'} \quad \leq \quad \texttt{a' -> Int -> a'}$$

and finally unifying to obtain the mgu:

$$[\texttt{b'} \mapsto \texttt{a'}, \texttt{c'} \mapsto \texttt{Int}]$$

5. Opening Existentials

Attempting to project from `intSet` directly would lead to a type error:

```
one = intSet.asList (intSet.add 1 intSet.empty)
error: cannot project "empty" from term of
       non-record type "exists f . Set Int f"
```

A variation of `let` explicitly "opens" any existentially quantified variables of the `let`-bound term:

```
one :: [Int]  {- inferred -}
one = let open s = intSet
      in s.asList (s.add 1 s.empty)
```