First-class Modules for Haskell

Mark Shields
Simon Peyton Jones
Microsoft Research Cambridge
mbs@cse.ogi.edu
simonpj@microsoft.com
Motivation

- Like ML, Haskell is stratified into module and core languages
- Unlike ML, Haskell's module language is just namespace control
- Signatures/Functors tend to be (badly!) emulated by classes/instances
  - eg: Edison datastructure library
- We could replace Haskell's module language with ML's, but not clear how to resolve overlap between functors and classes
  - Is a class a functor with implicit type parameterisation?
- However, unlike ML, Haskell's core language has good support for first-class ∀ and ∃ types
- Russo has given a (lightweight) semantics for ML modules by translation into ordinary ∀ and ∃ types
- So why not just encode ML signatures/functors as Haskell records/functions?
Motivation

- Problem 1: Haskell records are very weak
  - Field names must be unique
  - Projection doesn't use dot-notation
  - No nesting of type declarations
- Problem 2: Existentials must be wrapped by data constructor, unwrapped by case
  - Leads to many bogus datatypes
  - Can't share abstract types between modules since no common scope within which to place case
- Our work addresses these problems with 4 extensions to Haskell's type system
- Together support first-class modules with generative functors and recursive modules
Extension 1: Nominal Records

- "Using Parameterised Signatures to Express Modular Structure" [Jones, POPL'96]

```haskell
record Set a f = {
  empty :: f a
  add :: a -> f a -> f a
  asList :: f a -> [a]
}
intListSet :: Set Int []
intListSet = Set {
  empty = []
  add = \x xs -> x : filter (/= x) xs
  asList = id
}

Fields accessed by projection "."

one :: [Int]
one = intListSet.asList
  (intListSet.add 1 intListSet.empty)
```
Extension 1: Nominal Records

- Records are nominal
  - much simpler 😊
  - extends nicely to nominal subtyping
- Higher-kindred type arguments ok (of course!)
- May share field names, and fields may be polymorphic ...
- ... but require enough type annotations to determine record types

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

singleton :: forall m . Monad m -> m Int 
singleton m = m.fmap (+1) (m.unit 1)
```
Extension 1: Nominal Records

- Following [Odersky, Zenger, FOOL 8], records may contain nested type declarations, accessed by type projection "^".

```haskell
record BTSet a = {
    data BinTree = Leaf | Node BinTree a BinTree
    empty :: BinTree
    add :: a -> BinTree -> (BTSet a)^BinTree
}
```

- We can always \(\Lambda\)-lift to:

```haskell
data BTSet_BinTree a = BTSet_Leaf
                    | BTSet_Node (BTSet_BinTree a) a (BTSet_BinTree a)
record BTSet a = {
    empty :: BTSet_BinTree a
    add :: a -> BTSet_BinTree a -> BTSet_BinTree a
}
```

- **BUT** rather than combine type abstraction with records, we wish to use explicit existential types.

---

the same types
Extension 2: First-class Polymorphism

- "Putting Type Annotations to Work" [Odersky, Laufer, POPL'96] with a few extensions:
  - Constraints
    \[(\forall a . \text{Eq } a \Rightarrow a \rightarrow \text{Bool}) \rightarrow (\text{Int}, \text{Char}) \rightarrow \text{Bool}\]
  - Existentials
    \[(\forall a . a \rightarrow \exists b . (b, b \rightarrow a))\]
  - Annotation propagation
    \[f :: (\forall a . a \rightarrow a) \rightarrow (\text{Int}, \text{Char}) \rightarrow (\text{Int}, \text{Char})\]
    \[f \ g \ (x, y) = (g \ x, \ g \ y)\]
    (cf local/colored type inference)
  - Maintain universals in canonical prenex form
    (to avoid needless intermediate generalisations)

- (None of this is properly explained in the paper - we'll write it up as a stand-alone paper soon...)}
Extension 2: First-class Polymorphism

- Write type signatures for "functors" (functions) directly

```haskell
record Eq a = { eq :: a -> a -> Bool }
mkListSet :: forall a . Eq a -> exists f . Set a f
mkListSet eq = Set {
  empty = []
  add = \x xs -> x : filter (\y -> not (eq.eq x y)) xs
  asList = id
}

"mkListSet generates a Set from any Eq, and each such Set has a distinct and abstract implementation type"
```

- Application of functor yields something of existential type

```haskell
intSet :: exists f . Set Int f
intSet = mkListSet intEq
```

- Can choose between explicit and implicit parameterisation

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

- **BUT** we can't do anything with `intSet` – no destructor
Extension 3: Opened Bindings

- "Types for Modules" [Russo, PhD]
- Need a way to "open" existential quantifier independently of any data constructor
- New form of let binding
  
  ```
  let open s = mkListSet intEq
  in s.asList (s.add 1 s.empty)
  ```

- Any existential type vars are skolemized in let body
- Skolemized constants cannot escape
  
  ```
  let open s = mkListSet intEq
  in s.add 1 s.empty
  type error
  ```

Otherwise system is unsound

```
let f = \x -> let open y =
  ((x, (== x)) :: exists a . (a, a -> Bool)) in y
in (snd (f 1)) (fst (f True)) -- Crash!
```
Extension 3: Opened Bindings

- Each open introduces fresh skolem constants (i.e. "generative" rather than "applicative")
  
  ```ml
  let open s = mkListSet intEq
  in let open s' = mkListSet intEq
  in s.asList (s'.add 1 s.empty)
  type error
  ```

- Opened bindings may have type signatures
  
  ```ml
  let open s :: exists f . Set Int f
  open s = mkListSet intEq
  in s.asList (s.add 1 s.empty)
  ```

- **BUT** now consider variation
  
  ```ml
  let open s = mkListSet intEq
  in let t = s.empty
  in s.asList (s.add 1 t)
  ```

  How can *programmer* give a signature to `t` if *compiler* has chosen the skolemized type constant for `s`?
"Nested Types" [Odersky, Zenger, FOOL 8] (but underlying type-theoretic machinery very different)

Write \( x! \) to denote the type of
\[
\lambda x \ (y :: x!) \rightarrow (x, y) :: \forall a . a \rightarrow a \rightarrow (a, a)
\]

Write \( t^a \) to denote the binding of parameter \( a \) in type \( t \)
\[
(\text{Set \ Int } [])^f \text{Int} == [\text{Int}] \quad \text{-- surprised?}
\]
\[
(\text{Int, Bool})^\#1 == \text{Int}
\]
\[
(\text{Int} \rightarrow \text{Bool})^\text{arg} == \text{Int}
\]

Notice the type \( m!^t \) is reminiscent of the OCaml type \( M.t \)
(indeed, following Oderksy et al., we allow it to be written \( m.t \))

Of course alpha-conversion of all type arguments is no longer local (just as for record field names) ... 
- better to force programmer to use nested type synonym definition?
- or introduce type records (and record kinds)?
Extension 4: ! and ^

- Now we have a way to refer to a skolomized type without having to know any skolem constants
  ```haskell
  let open s :: exists f . Set Int f
      open s = mkListSet intEq
  in let t :: s.f Int
         t = s.empty
  in s.asList (s.add 1 t)
  ```
- Very useful for top-level bindings
Alas, Haskell programmer's seem to dislike ! and ^ 😞

Possible alternative

```haskell
let free f'    -- f' is fresh type variable
    open s :: exists f . Set Int f
    open s :: Set Int f' = mkListSet intEq
in let t :: f' Int
    t = s.empty
    in s.asList (t.add 1 t)
```

But to check that f' does not escape we must search the entire term under free (in addition to the context and result type)
record SetHelp a f = {
    addAll :: f a -> [a] -> f a
}

mkSetHelp :: forall a f . Set a f -> SetHelp a f
mkSetHelp set = SetHelp {
    addAll = foldr (set.add)
}

open intSet :: exists f . Set Int f
open intSet = mkListSet intEq

setHelp :: SetHelp Int (intSet.f)
setHelp = mkSetHelp intSet

two :: [Int]
two = intSet.asList
    (setHelp.addAll [1, 2] intSet.empty)
Top-level Interfaces and Implementations

- We split Haskell top-level modules into interfaces and implementations.
- Interfaces (record type bodies) live in ".hsi" files.
- Implementations (record bodies) live in ".hs" files.
- `open` is allowed at top-level (both signatures and bindings).
  - ok since nowhere for skolemized constants to escape to.
- Instance declarations split into declaration and binding.
Top-level Interfaces and Implementations

```haskell
record Lists = {
  data List a = Nil | Cons a (List a)
  map :: forall a . (a -> b) -> List a -> List b
  open set :: exists f . forall a . Eq a -> Set a f
  instance eqList :: Eq a => Eq (List a)
}

record Lists = {
  map = ...
  open set = \eq -> Set {
    empty = []
    ...
  }
  instance eqList = ...
}
```
Haskell Classes and Modules

- Classes are type-indexed values populated by instance definitions
- Classes may be at a deep level, instances must be at top-level
- record Eq a = {
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
}

class Eq a where Eq a       -- punning ok!

instance eqInt :: Eq Int     -- instance declaration
instance eqInt = Eq {
  (==) = intEq
  (/=) = \x y -> not (intEq x y)
}

(==) = ?Eq.(==)            -- replicate Haskell
(/=) = ?Eq.(/=)
Also...

- Mutually recursive bindings with abstract types
- Mutually recursive interfaces
Future Work?

- Could extend records with nominal subtyping...
  - ... a little ugly when combined with classes
  - ... but most details already worked out (see BABEL '01)

- Leads to interesting (useful?) encoding of OO-like features

  - Interface $\Rightarrow$ Record Declaration + Class Declaration
  - Class $\Rightarrow$ Instance Declaration + Functor
  - Object $\Rightarrow$ Value of abstract type
  - Subtyping $\Rightarrow$ Constraint Entailment
  - Inheritance $\Rightarrow$ Nominal Record Subtyping
  - Virtual Dispatch $\Rightarrow$ Overloading
  - Object Reference $\Rightarrow$ Data constructor with existential type

- Worth supporting with sugar?
Conclusions

- Nothing new, just careful combination of known systems
- Formalized
  - type checking (abstractly)
  - type inference and type-directed translation (in Haskell)
- Still to show usual type soundness, soundness & completeness of inference, type abstraction results
- Hopefully Simon will add to GHC...
  - ... probably without nested types and mutually recursive existentials
  - ... so far only higher-ranked polymorphism has made it in

- Paper: http://www.cse.ogi.edu/~mbs/pub/
- Compiler hx: http://www.haskell.org/~ghc/ (cvs)