Embedding effect systems in Haskell

@dorchard @tomaspetricek
Dominic Orchard & Tomas Petricek

Haskell Symposium, Thursday 4th September, Gothenburg

cabal install ixmonad
Motivation

We want to program with effects

.... to use different effects at the same time

.... to understand where effects happen

.... to understand which effects happen
Use monads?

```haskell
hello :: Monad m =>
    StateT String (StateT String m) ()

hello = do
    name ← get
    buff ← lift $ get
    lift $ put (buff ++ "hi! " ++ name)
```

- Not easily composed (see transformers above)
- Information is low (binary: pure or effectful)
Use monads?

```haskell
hello :: Monad m =>
    StateT String (StateT String m) ()

hello = do
    name ← get
    buff ← lift $ get
    lift $ put (buff ++ "hi! " ++ name)
```

- Not easily composed (see transformers above)
- Information is low (binary: pure or effectful)
Embed effect systems into (monadic) types

- more information
- aids composition: removes need for lifting
Technique

Classical effect systems [e.g. Gifford & Lucassen, 1986]

\[ \Gamma \vdash e : \tau, \ F \]

\[ \Gamma \vdash \text{put } y; \text{get } x: \tau, \ \{\text{Read}(x), \text{Write}(y)\} \]

\[ \Delta \vdash e_1 : a, \ F \quad \Delta, x: a \vdash e_2 : b, \ G \]

\[ \Delta \vdash \text{let } x = e_1 \text{ in } e_2 : b, \ F \sqcup G \]

\[ \frac{x : a \in \Gamma}{\Delta \vdash \var{\text{var}} x : a, \ \emptyset} \quad \frac{\Delta \vdash e : a, \ F \quad F \sqsubseteq G}{\Delta \vdash e : a, \ G} \]

\[ \Delta \vdash e : a, \ F \quad \Delta \vdash e : a, \ F \sqsubseteq G \]

\[ \Delta \vdash e : a, \ F \quad \Delta \vdash e : a, \ G \]

\[ \Delta \vdash e : a, \ F \quad \Delta \vdash e : a, \ G \]
Technique

Marry effects to monads [Wadler & Thiemann, 2003]

\[
\begin{array}{c}
\Gamma \vdash e_1 : M F a \quad \Gamma, x : a \vdash e_2 : M G b \\
\hline
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : M (F \sqcup G) b
\end{array}
\]

\[
\begin{array}{c}
x : a \in \Gamma \\
\hline
\Gamma \vdash x : M \emptyset a
\end{array}
\quad
\begin{array}{c}
\Gamma \vdash e : M F a \\
\hline
\Gamma \vdash e : M G a
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash e : M F a \\
F \subseteq G
\hline
\Gamma \vdash e : M G a
\end{array}
\]
Technique

Marry effects to monads semantically via parametric effect monads [Katsumata 2014]
also called indexed monads [Orchard, Petricek, Mycroft 2014]

\[ e : m F \tau \]

monoid \((F, \sqcup, \emptyset) + \sqsubseteq\)

+ epic GHC type system features
  = type-embedded classical effect systems
Parametric effect monads

Control.Effect

class Effect (m :: k → * → *) where
  type Unit m :: k
  type Plus m s t :: k

  return :: a → m (Unit m) a
  (>>=) :: m s a → (a → m t b) → m (Plus m s t) b

  (m i a) is not necessarily a monad

class Subeffect (m :: k → * → *) f g where
  sub :: m f a → m g a
(return \( x \)) >>= f
\equiv f \ x
\equiv m >>= \text{return}
\equiv m
\equiv m >>= (\lambda x \rightarrow (f \ x) >>= g)
\equiv (m >>= f) >>= g

\((k, \text{Unit } m, \text{Plus } m)\) is a monoid
Example 1: Reader effects

(ask x; ... ask y; ... ask z; ...) ::
  Reader {x :→ A, y :→ B, z :→ C} t

Effect sets of variable-type pairs

Variable-type pairs (mappings) :
  :: Symbol → * → *

Variables :
  Var :: Symbol → *
  e.g. Var :: Var “name”
Problem: type-level sets?

- Unordered container without duplicates
- Our approach:
  - type-level lists of pairs \( \text{"v" :} \, \mathbf{t} \)
  - normalise by sorting based on symbols
  - removing duplicates

- Uses \textit{data kinds}\(^1\) & \textit{closed types families}\(^2\)

\(^1\)[Yorgey, Weirich, Cretin, Peyton Jones, Vytiniotis, Magalhaes, 2012]
\(^2\)[Eisenberg, Vytiniotis, Peyton Jones, Weirich, 2014]
Type-level sets

data Set (n :: [*]) where
    Empty :: Set '[]
    Ext :: e \rightarrow Set s \rightarrow Set (e ': s)

type Union s t = Nub (Sort (Append s t))

kind of lists of types

bubble sort based on Symbols "v" in "v" \rightarrow t

type family Nub t where
    Nub '[] = '[]
    Nub '[e] = '[e]
    Nub (e ': e ': s) = Nub (e ': s)
    Nub (e ': f ': s) = e ': Nub (f ': s)
ask :: Var v \rightarrow R '[v :\rightarrow a] a

foo :: R '["name" :\rightarrow String] String
foo = do x \leftarrow ask (Var:::(Var "name"))
       return ("Name " ++ x)

bar :: Show a \Rightarrow R '["age" :\rightarrow a, "name" :\rightarrow String] String
bar = do x \leftarrow ask (Var:::(Var "name"))
      y \leftarrow ask (Var:::(Var "age"))
      return ("Name " ++ x ++ ". Age " ++ (show y))

*Main> runReader bar (Ext (Var :-> "Dom") (Ext (Var :-> 28) Empty))
"Name Dom. Age 28"
instance Effect (→) where

  type Unit (→) = '[]

  type Plus (→) s t = Union s t

return :: a → (Empty → a)

return x = Empty -> x

(>>=) :: (s → a) → (a → (t → b)) → (Union s t → b)

  e >>= k = let (s, t) = split st
             in  (k (e s)) t

split :: (Union s t) → (s, t)

ask :: Var v → ('[v :→ a] → a)

ask Var = Ext (Var :→ a) Empty :→ a
Example 2: Counter

```
data Counter (n :: Nat) a = Counter { forget :: a }

instance Effect Counter where
  type Unit Counter = 0
  type Plus Counter n m = n + m

  return :: a -> (Counter 0 a)
  return x = Counter x

  (>>=) :: Counter n a -> (a -> Counter m b) -> Counter (n + m) b
  (Counter a) >>= k = Counter . forget $ k a

  tick :: a -> Counter 1 a
  tick x = Counter x
```

[Danielsson 2008]
Example 2: Counter

verify complexity of map

map :: (a → Counter t b) →
       Vector n a → Counter (n * t) (Vector n b)
map f Nil       = return Nil
map f (Cons x xs) = do y ← f x
                        ys ← map f xs
                        return (Cons y ys)
Examples in the paper

<table>
<thead>
<tr>
<th></th>
<th>m :: k → * → *</th>
<th>k</th>
<th>Unit m :: k</th>
<th>Plus m :: k → k → k</th>
<th>Sub m :: k → k → Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>[Symbol → *]</td>
<td>‘[]</td>
<td>U</td>
<td></td>
<td>⊆</td>
</tr>
<tr>
<td>write</td>
<td>[Symbol → *]</td>
<td>‘[]</td>
<td>U</td>
<td></td>
<td>⊆</td>
</tr>
<tr>
<td>update</td>
<td>Maybe *</td>
<td>Nothing</td>
<td>V</td>
<td>Sub Nothing Just</td>
<td></td>
</tr>
<tr>
<td>state</td>
<td>[Symbol → * :! Eff]</td>
<td>‘[]</td>
<td>U*</td>
<td></td>
<td>⊆</td>
</tr>
<tr>
<td>counter</td>
<td>Nat</td>
<td>0</td>
<td>+</td>
<td></td>
<td>≤</td>
</tr>
<tr>
<td>array reader</td>
<td>[Sign Nat]</td>
<td>‘[]</td>
<td>U</td>
<td></td>
<td>⊇</td>
</tr>
</tbody>
</table>

data Eff = R | W | RW
Example 3: state (briefly)

get :: Var v → State '[v ↦ a :: R] a

put :: Var v → a → State '[v ↦ a :: W] ()

type family Nub t where
  Nub '[] = '[]
  Nub '[e] = '[e]
  Nub (e ': e ': as) = Nub (e ': as)
  Nub ((k ↦ a :: s) ': (k ↦ a :: t) ': as) =
    Nub ((k ↦ a :: RW) ': as)
  Nub (e ': f ': as) = e ': Nub (f ': as)
Example 3: state (briefly)

```
get :: Var v → State '[v :→ a :! R] a
put :: Var v → a → State '[v :→ a :! W] ()
```

type family Nub t where

\[
\begin{align*}
\text{Nub} & \ '[] \ ' = '[] \\
\text{Nub} & \ ' [e] \ = ' [e] \\
\text{Nub} \ (e \ ': \ e \ ': \text{as}) & = \text{Nub} \ (e \ ': \text{as}) \\
\text{Nub} \ ((k \ :\to \ a \ :! \ s) \ ': (k \ :\to \ a \ :! \ t) \ ': \text{as}) & = \\
& \quad \text{Nub} \ ((k \ :\to \ a \ :! \ RW) \ ': \text{as}) \\
\text{Nub} \ (e \ ': \ f \ ': \text{as}) & = e \ ': \text{Nub} \ (f \ ': \text{as})
\end{align*}
\]
Also in the paper

- Lots of examples
- Effect polymorphism
- **Coeffects** and implicit parameters

  implicit parameters = coeffect system!

  [can couple coeffects with codo notation]

- All the details of type/value-level sets
- Subeffecting
Compositionality & generality

- An alternate approach to combining effects

```haskell
class Monad m => Put a m where put :: a -> m ()
class Monad m => Get a m where get :: m a
```

- Constraints are sets

- But less general (parametric effect monads parameterised by arbitrary monoid)
Concluding thoughts I

- Intermediate between monads & effect handlers
- Could use as an effect system for handlers
  e.g. for [Kammar, Lindley, Oury, ICFP13]
- No need for language extensions / macros
  - Embeds easily with existing monadic approach
Concluding thoughts 2

- GHC types very rich but still lots of cruft
- Sometimes extra signatures needed :/
- Native type-level sets would be nice!
Thanks!

cabal install ixmonad
http://github.com/dorchard/ixmonad

Summary:

Parametrisable effect system for the do-notation embedded into the types via parametric effect monads

@dorchard @tomaspetricek