Embedding effect systems in Haskell

@dorchard @tomaspetricek

Dominic Orchard & Tomas Petricek

UNIVERSITY OF CAMBRIDGE

Haskell Symposium, Thursday 4th September, Gothenburg

`cabal install ixmonad`

`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?

```
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?

```
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)

this work….

```
hello :: State
        '["buff" :→ String :! RW, "name" :→ String :! R] ()

hello = do  name ← get Var::(Var "name")
            buff ← get Var::(Var "buff")
            put Var::(Var "buff")(buff ++ "hi! " ++ name)
```

- Embed effect systems into (monadic) types
  - more information
  - aids composition: removes need for lifting

Technique

$$\Gamma \vdash e : \tau, \mathsf{F}$$

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

$$\Gamma \vdash \texttt{put}\ y;\ \texttt{get}\ x : \tau, \{\mathrm{Read}(x), \mathrm{Write}(y)\}$$

$$\frac{\Gamma \vdash e_1 : a, \mathsf{F} \qquad \Gamma, x : a \vdash e_2 : b, \mathsf{G}}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : b, \mathsf{F} \sqcup \mathsf{G}}\ \text{let}$$

$$\frac{x : a \in \Gamma}{\Gamma \vdash x : a, \emptyset}\ \text{var} \qquad \frac{\Gamma \vdash e : a, \mathsf{F} \qquad \mathsf{F} \sqsubseteq \mathsf{G}}{\Gamma \vdash e : a, \mathsf{G}}\ \text{sub}$$

Technique

$e : m \; \mathsf{F} \; \tau$

Marry effects to monads [Wadler & Thiemann, 2003]

$$\frac{\Gamma \vdash e_1 : \mathsf{M}\;\mathsf{F}\;a \quad \Gamma, x : a \vdash e_2 : \mathsf{M}\;\mathsf{G}\;b}{\Gamma \vdash \texttt{let}\; x = e_1 \;\texttt{in}\; e_2 : \mathsf{M}\;(\mathsf{F} \sqcup \mathsf{G})\;b}\;\text{let}$$

$$\frac{x : a \in \Gamma}{\Gamma \vdash x : \mathsf{M}\;\varnothing\;a}\;\text{var} \qquad \frac{\Gamma \vdash e : \mathsf{M}\;\mathsf{F}\;a \quad \mathsf{F} \sqsubseteq \mathsf{G}}{\Gamma \vdash e : \mathsf{M}\;\mathsf{G}\;a}\;\text{sub}$$

Technique

Marry effects to monads **semantically** via

parametric effect monads [Katsumata 2014]

also called *indexed monads* [Orchard, Petricek, Mycroft 2014]

$$e : m\ \mathsf{F}\ \tau$$

monoid $(\mathsf{F}, \sqcup, \emptyset) + \sqsubseteq$

+ 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
```

$$
\begin{aligned}
& (\mathit{return}\ x) \gg\!= f \\
\equiv\quad & f\ x
\end{aligned}
$$

$$
\begin{aligned}
& m \gg\!= \mathit{return} \\
\equiv\quad & m
\end{aligned}
$$

$$
\begin{aligned}
& m \gg\!= (\lambda x \rightarrow (f\ x) \gg\!= g) \\
\equiv\quad & (m \gg\!= f) \gg\!= g
\end{aligned}
$$

(`k`, `Unit m`, `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 `"v" :→ t`
  - normalise by sorting based on symbols
  - removing duplicates

- Uses *data kinds*[1] & *closed types families*[2]

[1][Yorgey, Weirich, Cretin, Peyton Jones, Vytiniotis, Magalhaes, 2012]
[2][Eisenberg, Vytiniotis, Peyton Jones, Weirich, 2014]

Type-level sets

kind of lists of types

```
data Set (n :: [*]) where
     Empty :: Set '[]
     Ext :: e → Set s → Set (e ': s)
```

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

bubble sort based on Symbols "v" in "v" :→ 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 → R '[v :→ a] a
```

```
foo :: R '[“name” :→ String] String
foo = do x ← ask (Var::(Var “name”))
         return ("Name " ++ x)
```

```
bar :: Show a => R '[“age” :→ a, “name” :→ String] String
bar = do x  ← ask (Var::(Var “name”))
         y  ← 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  = \st → let (s, t) = split st
                  in  (k (e s)) t
```

```
Reader r a = r → a
```

```
split :: (Union s t) → (s, t)
```

```
ask :: Var v → ('[v :→ a] → a)
ask Var = \(Ext (Var :-> a) Empty) → a
```

Example 2: *Counter*

Control.Effect.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

	k	Unit m :: k	Plus m :: k → k → k	Sub m :: k → k → Constraint
m :: k → * → *				
read	[Symbol :→ *]	'[]	$\cup$	$\subseteq$
write	[Symbol :→ *]	'[]	$\cup$	$\subseteq$
update	Maybe *	Nothing	$\vee$	Sub Nothing Just
state	[Symbol :→ * :! Eff]	'[]	$\cup^*$	$\subseteq$
counter	Nat	0	+	$\leq$
array reader	[Sign Nat]	'[]	$\cup$	$\supseteq$

```
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
    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)
```

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

```
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 1

- 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!

ICFP 2015?!

ICFP 2015?!

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