# Embedding effect systems in Haskell

@dorchard @tomaspetricek

Dominic Orchard & Tomas Petricek



Haskell Symposium, Thursday 4th September, Gothenburg

### cabal install ixmonad

### Motivation

#### cabal install ixmonad

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)
```

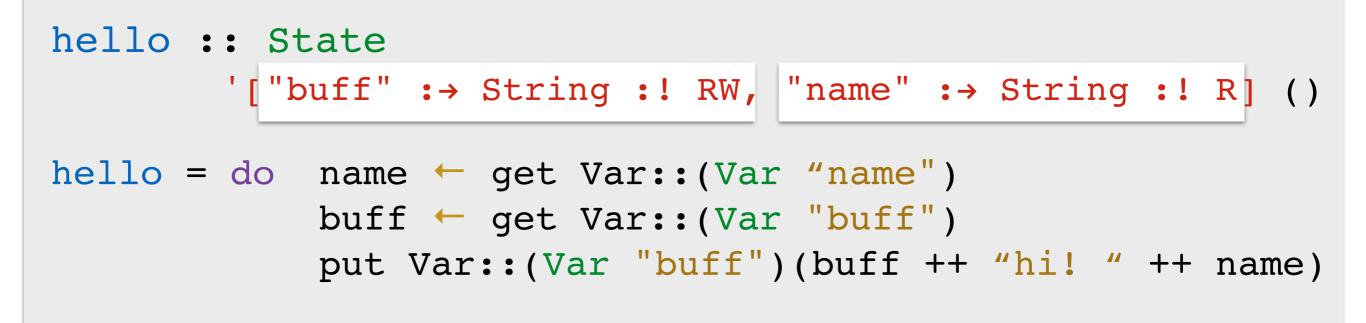
- Not easily composed (see transformers above)
- Information is low (binary: pure or effectful)

### 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....



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



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

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

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

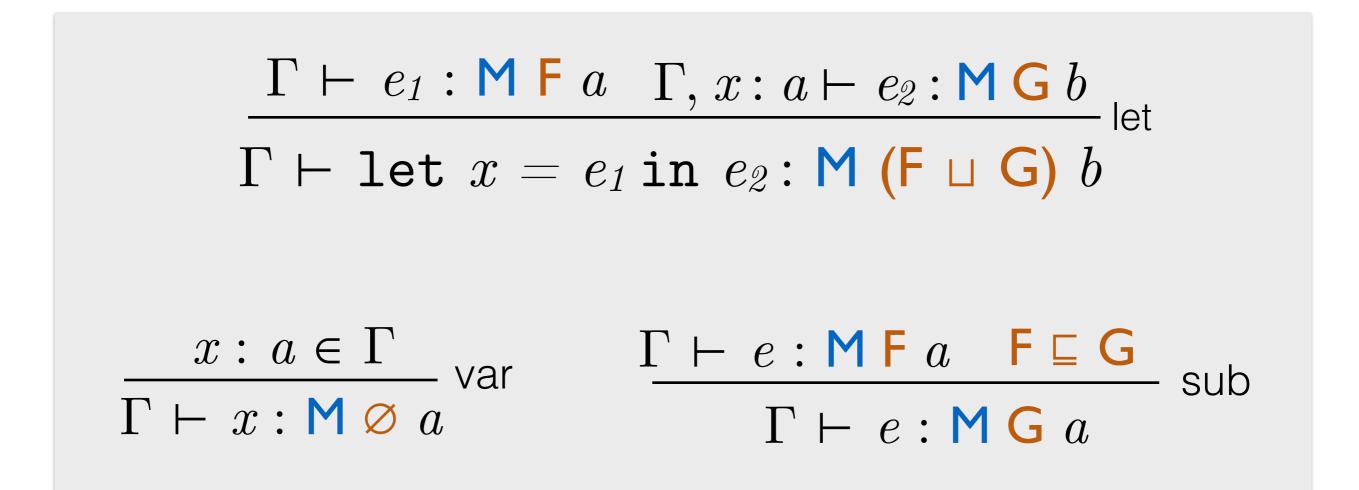
$$\frac{\Gamma \vdash e_{1} : a, \mathsf{F}}{\Gamma \vdash \mathsf{let} \ x = e_{1} \text{ in } e_{2} : b, \mathsf{F} \sqcup \mathsf{G}} \text{ let}$$

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





### Marry effects to monads [Wadler & Thiemann, 2003]





# Marry effects to monads semantically via parametric effect monads [Katsumata 2014]

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

$$e:m \mathsf{F} au$$
  
monoid  $(\mathsf{F}, \sqcup, \varnothing) + \sqsubseteq$ 

### + 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 \rightarrow m (Unit m) a
(>>=) :: m s a \rightarrow (a \rightarrow m t b) \rightarrow m (Plus m s t) b
```

(m i a) is not necessarily a monad

class Subeffect (m ::  $k \rightarrow * \rightarrow *$ ) f g where sub :: m f a  $\rightarrow$  m g a

$$= \begin{array}{ccc} (return \ x) \gg f \\ \equiv f \ x \\ m \gg return \\ \equiv m \\ m \gg (\lambda x \rightarrow (f \ x) \gg g \\ \equiv (m \gg f) \gg g \end{array}$$

### (k, Unit m, Plus m) is a monoid

### Example I: Reader effects

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

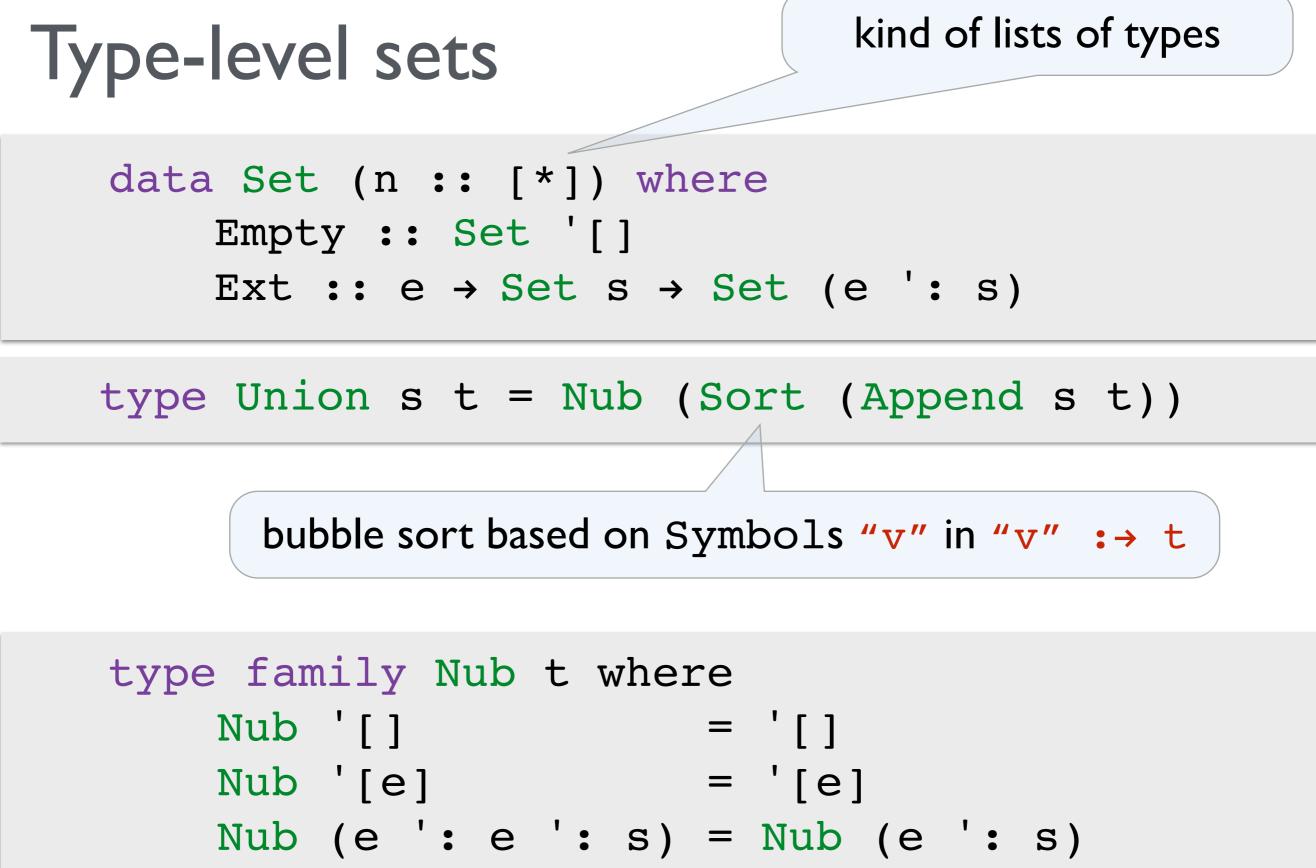
### Effect sets of variable-type pairs

Variable-type pairs (mappings)	$: \rightarrow :: Symbol \rightarrow * \rightarrow *$	
Variables	Var :: Symbol $\rightarrow$ *	
	<i>e.g.</i> Var <b>::</b> Var "name"	

# Problem: type-level sets?

- Unordered container without duplicates
- Our approach:
  - by type-level lists of pairs "v" :→ t
  - normalise by sorting based on symbols
  - removing duplicates
- Uses <u>data kinds</u><sup>1</sup> & <u>closed types families</u><sup>2</sup>

<sup>1</sup>[Yorgey, Weirich, Cretin, Peyton Jones, Vytiniotis, Magalhaes, 2012] <sup>2</sup>[Eisenberg, Vytiniotis, Peyton Jones, Weirich, 2014]



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

\*Main> runReader bar (Ext (Var :-> "Dom") (Ext (Var :-> 28) Empty))
"Name Dom. Age 28"

#### Control.Effect.Reader

instance Effect  $(\rightarrow)$  where Reader  $r a = r \rightarrow a$ type Unit  $(\rightarrow) = '[]$ type Plus  $(\rightarrow)$  s t = Union s t return ::  $a \rightarrow (Empty \rightarrow a)$ return  $x = \mathbb{E}mpty \rightarrow x$ (>>=) ::  $(s \rightarrow a) \rightarrow (a \rightarrow (t \rightarrow b)) \rightarrow (Union s t \rightarrow b)$  $e >>= k = \st \rightarrow let (s, t) = split st$ in (k (e s)) t split :: (Union s t)  $\rightarrow$  (s, t) ask :: Var  $v \rightarrow ('[v : \rightarrow a] \rightarrow a)$ ask Var =  $(Ext (Var : -> a) Empty) \rightarrow a$ 

# 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 \rightarrow (Counter 0 a)
```

```
return x = Counter x
```

```
(>>=) :: Counter n a \rightarrow (a \rightarrow Counter m b) \rightarrow Counter (n + m) b
(Counter a) >>= k = Counter . forget $ k a
```

```
tick :: a \rightarrow 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

m :: k → * → *	k	Unit m :: k	Plus m :: $k \rightarrow k \rightarrow k$	Sub m :: k → k → Constraint
read	[Symbol :→ *]	'[]	U	C
write	[Symbol :→ *]	'[]	U	$\subseteq$
update	Maybe *	Nothing	V	Sub Nothing Just
state	[Symbol :→ * :! Eff]	'[]	U*	C
counter	Nat	0	+	$\leq$
array reader	[Sign Nat]	'[]	U	⊇

data Eff =  $R \mid W \mid RW$ 

# Example 3: state (briefly)

get :: Var  $v \rightarrow State '[v : \rightarrow a :! R] a$ put :: Var  $v \rightarrow a \rightarrow State '[v : \rightarrow a :! W] ()$ 

```
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  $v \rightarrow State '[v : \rightarrow a :! R] a$ put :: Var  $v \rightarrow a \rightarrow State '[v : \rightarrow a :! W] ()$ 

# Also in the paper

- Lots of examples
- Effect polymorphism
- <u>Coeffects</u> 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 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 donotation embedded into the types via parametric effect monads

@dorchard @tomaspetricek