

Free Theorems about Monadic Code

Janis Voigtländer

University of Bonn

EWCE'11

Functional Programming and Reasoning

Goodies of (pure) FP:

- ▶ declarative
- ▶ abstraction/modularity
- ▶ referential transparency

Functional Programming and Reasoning

Goodies of (pure) FP:

- ▶ declarative
- ▶ abstraction/modularity
- ▶ referential transparency

Methods for analysing/verifying programs:

- ▶ equational reasoning
- ▶ algebraic and logical techniques
- ▶ type-based reasoning

Programming with Side Effects (in Haskell)

Example:

```
echo :: IO ()  
echo = do c ← getChar  
         when (c ≠ '*') $  
           do putChar c  
              echo
```

Programming with Side Effects (in Haskell)

Example:

```
echo :: IO ()  
echo = do c ← getChar  
        when (c ≠ '*') $  
            do putChar c  
                echo
```

Essence:

- ▶ program in imperative style where wanted

Programming with Side Effects (in Haskell)

Example:

```
echo :: IO ()
echo = do c ← getChar
         when (c ≠ '*') $
           do putChar c
              echo
```

Essence:

- ▶ program in imperative style where wanted
- ▶ ..., and only there!

Programming with Side Effects (in Haskell)

Example:

```
echo :: IO ()  
echo = do c ← getChar  
        when (c ≠ '*') $  
            do putChar c  
                echo
```

Essence:

- ▶ program in imperative style where wanted
- ▶ ..., and only there!
- ▶ type system ensures separation

Programming with Side Effects (in Haskell)

Example:

```
echo :: IO ()
echo = do c ← getChar
        when (c ≠ '*') $
          do putChar c
            echo
```

Essence:

- ▶ program in imperative style where wanted
- ▶ ..., and only there!
- ▶ type system ensures separation
- ▶ abstraction mechanisms fully available

Programming with Side Effects (in Haskell)

Example:

```
echo :: IO ()
echo = do c ← getChar
         when (c ≠ '*') $
           do putChar c
              echo
```

Essence:

- ▶ program in imperative style where wanted
- ▶ ..., and only there!
- ▶ type system ensures separation
- ▶ abstraction mechanisms fully available

But: formal reasoning techniques?

Papers (at the time)



A. Filinski and K. Støvring.

Inductive reasoning about effectful data types.

In International Conference on Functional Programming, Proceedings, pages 97–110. ACM Press, 2007.



G. Hutton and D. Fulger.

Reasoning about effects: Seeing the wood through the trees.

In Trends in Functional Programming, Draft Proceedings, 2008.



W. Swierstra and T. Altenkirch.

Beauty in the beast — A functional semantics for the awkward squad.

In Haskell Workshop, Proceedings, pages 25–36. ACM Press, 2007.

Free Theorems [Wadler '89]

For every function

$$g :: [\alpha] \rightarrow [\alpha]$$

it holds

$$\text{map } f (g l) = g (\text{map } f l)$$

for arbitrary f and l , where

$$\text{map} :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$$

$$\text{map } f [] = []$$

$$\text{map } f (a : as) = (f a) : (\text{map } f as)$$

Free Theorems [Wadler '89]

For every function

$$g :: [\alpha] \rightarrow [\alpha]$$

it holds

$$\text{map } f (g l) = g (\text{map } f l)$$

for arbitrary f and l , where

$$\begin{aligned} \text{map} &:: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\ \text{map } f [] &= [] \\ \text{map } f (a : as) &= (f a) : (\text{map } f as) \end{aligned}$$

Some applications:

- ▶ efficiency improving program transformations [Gill et al. '93]
- ▶ meta-theorems about classes of algorithms [V. '08a]
- ▶ solutions to the view-update problem [V. '09a]
- ▶ reducing testing effort [Bernardy et al. '10]

From Programming to Reasoning

From:

You must judge for yourself, but I believe that the monadic approach to programming, in which actions are first class values, is itself interesting, beautiful, and modular. In short, Haskell is the world's finest imperative programming language.

[Peyton Jones '01]

From Programming to Reasoning

From:

You must judge for yourself, but I believe that the monadic approach to programming, in which actions are first class values, is itself interesting, beautiful, and modular. In short, Haskell is the world's finest imperative programming language.

[Peyton Jones '01]

To:

Parametricity [Wadler '89] allows the derivation of theorems for a whole class of programs, only knowing their type. Voigtländer [V. '09b] has recently shown how to extend the parametricity approach to type constructor classes such as Monad. This way we can derive theorems about effectful programs without knowing the particular effects used.

[Oliveira et al. '10]

Monads in Haskell

Example 1:

```
echo :: IO ()
echo = do c ← getChar
         when (c ≠ '*') $
           do putChar c
              echo
```

Monads in Haskell

Example 1:

```
echo :: IO ()
echo = do c ← getChar
         when (c ≠ '*') $
           do putChar c
              echo
```

Example 2:

```
sequence :: Monad m => [m a] → m [a]
sequence [] = return []
sequence (m : ms) = do a ← m
                       as ← sequence ms
                       return (a : as)
```


Monads in Haskell

Example 1:

```
echo :: IO ()
echo = do c ← getChar
         when (c ≠ '*') $
           do putChar c
              echo
```

Effectful
operations!

Example 2:

```
sequence :: Monad m => [m a] → m [a]
sequence [] = return []
sequence (m : ms) = do a ← m
                       as ← sequence ms
                       return (a : as)
```

Monads in Haskell

Example 1:

A specific monad!

```
echo :: IO ()  
echo = do c ← getChar  
         when (c ≠ '*') $  
           do putChar c  
              echo
```

Effectful
operations!

Example 2:

```
sequence :: Monad m => [m a] → m [a]  
sequence [] = return []  
sequence (m : ms) = do a ← m  
                       as ← sequence ms  
                       return (a : as)
```

Monads in Haskell

Example 1:

A specific monad!

```
echo :: IO ()  
echo = do c ← getChar  
         when (c ≠ '*') $  
           do putChar c  
              echo
```

Effectful
operations!

Example 2:

Parametric over a monad!

```
sequence :: Monad m ⇒ [m a] → m [a]  
sequence [] = return []  
sequence (m : ms) = do a ← m  
                       as ← sequence ms  
                       return (a : as)
```

Monads in Haskell

Example 1:

A specific monad!

```
echo :: IO ()  
echo = do c ← getChar  
         when (c ≠ '*') $  
           do putChar c  
             echo
```

Effectful
operations!

Example 2:

Parametric over a monad!

```
sequence :: Monad m ⇒ [m a] → m [a]  
sequence [] = return []  
sequence (m : ms) = do a ← m  
                       as ← sequence ms  
                       return (a : as)
```

No specific
(new) effects!

Monads in Haskell

Example 2:

Parametric over a monad!

`sequence` :: `Monad m` \Rightarrow `[m a]` \rightarrow `m [a]`

`sequence []` = `return []`

`sequence (m : ms)` = **do** `a` \leftarrow `m`

`as` \leftarrow `sequence ms`

No specific
(new) effects!

`return (a : as)`

Monads in Haskell

Example 2:

Parametric over a monad!

`sequence` :: `Monad m` \Rightarrow `[m a]` \rightarrow `m [a]`

No specific
(new) effects!

A Slightly More Simple Example

`f` :: Monad *m* ⇒ *m a* → *m a* → *m a*

`f` *m*₁ *m*₂ =

A Slightly More Simple Example

$f :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

$f\ m_1\ m_2 = \mathbf{do}\ m_1$

A Slightly More Simple Example

```
f :: Monad m => m a -> m a -> m a
f m1 m2 = do m1
           a <- m1
```

A Slightly More Simple Example

$f :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

$f\ m_1\ m_2 = \mathbf{do}\ m_1$
 $a \leftarrow m_1$
 m_2

A Slightly More Simple Example

$f :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

```
f m1 m2 = do m1  
           a ← m1  
           m2  
           b ← m1
```

A Slightly More Simple Example

$f :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

```
f m1 m2 = do m1
           a ← m1
           m2
           b ← m1
           c ← m2
```

A Slightly More Simple Example

`f` :: Monad $m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

```
f m1 m2 = do m1  
             a ← m1  
             m2  
             b ← m1  
             c ← m2  
             return b
```

A Slightly More Simple Example

`f` :: Monad $m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

`f` $m_1\ m_2 = \mathbf{do}$ m_1

$a \leftarrow m_1$

m_2

$b \leftarrow m_1$

$c \leftarrow m_2$

`return` b

No effects
introduced!

A Slightly More Simple Example

`f` :: Monad $m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

`f` $m_1\ m_2 = \mathbf{do}\ m_1$

$a \leftarrow m_1$

m_2

$b \leftarrow m_1$

$c \leftarrow m_2$

`return` b

But m_1, m_2 may
encapsulate ones!

No effects
introduced!

A Slightly More Simple Example

Assume m_1, m_2 are pure.

```
f :: Monad m => m a -> m a -> m a
f m1 m2 = do m1
           a <- m1
           m2
           b <- m1
           c <- m2
           return b
```


A Slightly More Simple Example

Assume m_1, m_2 are pure.

That is, $m_1 = (\text{return } u)$ and $m_2 = (\text{return } v)$ for some u, v .

$f :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

```
f m1 m2 = do m1
            a ← m1
            m2
            b ← m1
            c ← m2
            return b
```

A Slightly More Simple Example

Assume m_1, m_2 are pure.

That is, $m_1 = (\text{return } u)$ and $m_2 = (\text{return } v)$ for some u, v .

Then:

$f :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

```
f m1 m2 = do return u
           a ← return u
           return v
           b ← return u
           c ← return v
           return b
```

A Slightly More Simple Example

Assume m_1, m_2 are pure.

That is, $m_1 = (\text{return } u)$ and $m_2 = (\text{return } v)$ for some u, v .

Then:

```
f :: Monad m => m a -> m a -> m a
```

```
f m1 m2 = do return u
```

```
    a <- return u
```

```
    return v
```

```
    b <- return u
```

```
    c <- return v
```

```
    return b
```

```
(return u) >> m = m
```

A Slightly More Simple Example

Assume m_1, m_2 are pure.

That is, $m_1 = (\text{return } u)$ and $m_2 = (\text{return } v)$ for some u, v .

Then:

$f :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

$f\ m_1\ m_2 = \text{do return } u$

$a \leftarrow \text{return } u$

$\text{return } v$

$b \leftarrow \text{return } u$

$c \leftarrow \text{return } v$

$\text{return } b$

$(\text{return } u) \gg m = m$

A Slightly More Simple Example

Assume m_1, m_2 are pure.

That is, $m_1 = (\text{return } u)$ and $m_2 = (\text{return } v)$ for some u, v .

Then:

$f :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

$f\ m_1\ m_2 = \mathbf{do}$

$a \leftarrow \text{return } u$

$\text{return } v$

$b \leftarrow \text{return } u$

$c \leftarrow \text{return } v$

$\text{return } b$

$(\text{return } u) \gg m = m$

A Slightly More Simple Example

Assume m_1, m_2 are pure.

That is, $m_1 = (\text{return } u)$ and $m_2 = (\text{return } v)$ for some u, v .

Then:

$f :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

$f\ m_1\ m_2 = \mathbf{do}$

$a \leftarrow \text{return } u$

$\text{return } v$

$b \leftarrow \text{return } u$

$c \leftarrow \text{return } v$

$\text{return } b$

$(\text{return } u) \gg= (\lambda a \rightarrow m) = m[u/a]$

A Slightly More Simple Example

Assume m_1, m_2 are pure.

That is, $m_1 = (\text{return } u)$ and $m_2 = (\text{return } v)$ for some u, v .

Then:

$f :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

$f\ m_1\ m_2 = \mathbf{do}$

$\text{return } v$

$b \leftarrow \text{return } u$

$c \leftarrow \text{return } v$

$\text{return } b$

$(\text{return } u) \gg= (\lambda a \rightarrow m) = m[u/a]$

A Slightly More Simple Example

Assume m_1, m_2 are pure.

That is, $m_1 = (\text{return } u)$ and $m_2 = (\text{return } v)$ for some u, v .

Then:

$f :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

$f\ m_1\ m_2 = \mathbf{do}$

$\text{return } v$

$b \leftarrow \text{return } u$

$c \leftarrow \text{return } v$

$\text{return } b$

$(\text{return } v) \gg m = m$

A Slightly More Simple Example

Assume m_1, m_2 are pure.

That is, $m_1 = (\text{return } u)$ and $m_2 = (\text{return } v)$ for some u, v .

Then:

```
f :: Monad m => m a -> m a -> m a
f m1 m2 = do
```

```
  b <- return u
  c <- return v
  return b
```

```
(return v) >> m = m
```

A Slightly More Simple Example

Assume m_1, m_2 are pure.

That is, $m_1 = (\text{return } u)$ and $m_2 = (\text{return } v)$ for some u, v .

Then:

$f :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

$f\ m_1\ m_2 = \mathbf{do}$

$b \leftarrow \text{return } u$

$c \leftarrow \text{return } v$

$\text{return } b$

$(\text{return } u) \gg= (\lambda b \rightarrow m) = m[u/b]$

A Slightly More Simple Example

Assume m_1, m_2 are pure.

That is, $m_1 = (\text{return } u)$ and $m_2 = (\text{return } v)$ for some u, v .

Then:

$f :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

$f\ m_1\ m_2 = \mathbf{do}$

$c \leftarrow \text{return } v$

$\text{return } u$

$(\text{return } u) \gg= (\lambda b \rightarrow m) = m[u/b]$

A Slightly More Simple Example

Assume m_1, m_2 are pure.

That is, $m_1 = (\text{return } u)$ and $m_2 = (\text{return } v)$ for some u, v .

Then:

$f :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

$f\ m_1\ m_2 = \mathbf{do}$

$c \leftarrow \text{return } v$

$\text{return } u$

$(\text{return } v) \gg= (\lambda c \rightarrow m) = m[v/c]$

A Slightly More Simple Example

Assume m_1, m_2 are pure.

That is, $m_1 = (\text{return } u)$ and $m_2 = (\text{return } v)$ for some u, v .

Then:

$f :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

$f\ m_1\ m_2 = \mathbf{do}$

$\text{return } u$

$(\text{return } v) \gg= (\lambda c \rightarrow m) = m[v/c]$

A Slightly More Simple Example

Assume m_1, m_2 are pure.

That is, $m_1 = (\text{return } u)$ and $m_2 = (\text{return } v)$ for some u, v .

Then:

$f :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

$f\ m_1\ m_2 = \mathbf{do}$

$\text{return } u$

Purity is propagated!

A Slightly More Simple Example

Assume m_1, m_2 are pure.

That is, $m_1 = (\text{return } u)$ and $m_2 = (\text{return } v)$ for some u, v .

Then:

```
f :: Monad m => m a -> m a -> m a
f m1 m2 = do
```

```
  return u
```

Purity is propagated!

What about other “invariants”?

Propagating Invariants

`f` :: Monad $m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

```
f m1 m2 = do m1  
             a ← m1  
             m2  
             b ← m1  
             c ← m2  
             return b
```


Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$,

$f :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$

```
f m1 m2 = do m1
           a ← m1
           m2
           b ← m1
           c ← m2
           return b
```

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_j = \text{id}$.

$f :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$

```
f m1 m2 = do m1
           a ← m1
           m2
           b ← m1
           c ← m2
           return b
```

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$

```
 $\text{f } m_1 m_2 = \text{do } m_1$   
     $a \leftarrow m_1$   
     $m_2$   
     $b \leftarrow m_1$   
     $c \leftarrow m_2$   
     $\text{return } b$ 
```

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$

$\text{f } m_1 m_2 = \text{do } m_1$

$a \leftarrow m_1$

m_2

$b \leftarrow m_1$

$c \leftarrow m_2$

$\text{return } b$

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 \ m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m \ a \rightarrow m \ a \rightarrow m \ a$

```
 $\text{f } m_1 \ m_2 = \text{do } m_1^S$   
     $a \leftarrow m_1$   
     $m_2$   
     $b \leftarrow m_1$   
     $c \leftarrow m_2$   
     $\text{return } b$ 
```

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$

```
 $\text{f } m_1 m_2 = \text{do } m_1^s$   
     $a \leftarrow m_1^s$   
     $m_2$   
     $b \leftarrow m_1$   
     $c \leftarrow m_2$   
     $\text{return } b$ 
```

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

```
f :: Monad m => m a -> m a -> m a
f m1 m2 = do Sm1 S
             a <- Sm1 S
             m2
             b <- m1
             c <- m2
             return b
```

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

```
f :: Monad m => m a -> m a -> m a
f m1 m2 = do Sm1 S
             a <- Sm1 S
             Sm2
             b <- m1
             c <- m2
             return b
```


Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

```
f :: Monad m => m a -> m a -> m a
f m1 m2 = do Sm1 S
             a <- Sm1 S
             Sm2 S
             b <- m1
             c <- m2
             return b
```

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

```
f :: Monad m => m a -> m a -> m a
f m1 m2 = do Sm1 S
             a <- Sm1 S
             Sm2 S
             b <- Sm1
             c <- m2
             return b
```

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 \ m_2) = \text{id}$?

```
f :: Monad m => m a -> m a -> m a
f m1 m2 = do Sm1 S
             a <- Sm1 S
             Sm2 S
             b <- Sm1 S
             c <- m2
             return b
```

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

```
f :: Monad m => m a -> m a -> m a
f m1 m2 = do Sm1 S
             a <- Sm1 S
             Sm2 S
             b <- Sm1 S
             c <- Sm2
             return b
```

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

```
f :: Monad m => m a -> m a -> m a
f m1 m2 = do Sm1 S
             a <- Sm1 S
             Sm2 S
             b <- Sm1 S
             c <- Sm2 S
             return b
```

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$

```
 $\text{f } m_1 m_2 = \text{do } m_1^S$   
     $a \leftarrow m_1^S$   
     $m_2^S$   
     $b \leftarrow m_1^S$   
     $c \leftarrow m_2^S$   
     $\text{return } b^S$ 
```

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$

```
 $\text{f } m_1 m_2 = \text{do } m_1^S$   
     $a \leftarrow m_1^S$   
     $m_2^S$   
     $b \leftarrow m_1^S$   
     $c \leftarrow m_2^S$   
     $\text{return } b^S$ 
```

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$

```
 $\text{f } m_1 m_2 = \text{do } m_1$   
     $a \leftarrow m_1$   
     $m_2$   
     $b \leftarrow m_1$   
     $c \leftarrow m_2$   
     $\text{return } b$ 
```


Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$

$\text{f } m_1 m_2 = \text{do } m_1$

$a \leftarrow m_1$

m_2

$b \leftarrow m_1$

$c \leftarrow m_2$

$\text{State } (\lambda s \rightarrow (b, s))$

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

```
f :: Monad m => m a -> m a -> m a
f m1 m2 = do m1
            a <- m1
            m2
            b <- m1
            c <- State (\s -> (... , s))
            State (\s -> (b, s))
```

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

```
f :: Monad m => m a -> m a -> m a
f m1 m2 = do m1
           a <- m1
           m2
           b <- m1
           c <- State (\s -> (... , s))
           State (\s -> (b, s))
```

```
(State (\s -> (... , s))) >>= (\c -> State (\s -> (b, s))) = ?
```

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$

$\text{f } m_1 m_2 = \text{do } m_1$

$a \leftarrow m_1$

m_2

$b \leftarrow m_1$

$\text{State } (\lambda s \rightarrow (b, s))$

$(\text{State } (\lambda s \rightarrow (\dots, s))) \gg= (\lambda c \rightarrow \text{State } (\lambda s \rightarrow (b, s))) = ?$

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$

$\text{f } m_1 m_2 = \mathbf{do} \ m_1$

$\ a \leftarrow m_1$

$\ m_2$

$\ b \leftarrow m_1$

$\ \text{State } (\lambda s \rightarrow (b, s))$

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

```
f :: Monad m => m a -> m a -> m a
f m1 m2 = do m1
            a <- m1
            m2
            b <- State (\s -> (... , s))
            State (\s -> (b, s))
```

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

```
f :: Monad m => m a -> m a -> m a
f m1 m2 = do m1
            a <- m1
            m2
            b <- State (\s -> (... , s))
            State (\s -> (b, s))
```

```
(State (\s -> (... , s))) >>= (\b -> State (\s -> (b, s))) = ?
```

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \ \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 \ m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m \ a \rightarrow m \ a \rightarrow m \ a$

$\text{f } m_1 \ m_2 = \text{do } m_1$

$\quad a \leftarrow m_1$

$\quad m_2$

$\quad \text{State } (\lambda s \rightarrow (\dots, s))$

$(\text{State } (\lambda s \rightarrow (\dots, s))) \gg= (\lambda b \rightarrow \text{State } (\lambda s \rightarrow (b, s))) = ?$

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$

$\text{f } m_1 m_2 = \text{do } m_1$

$a \leftarrow m_1$

m_2

$\text{State } (\lambda s \rightarrow (\dots, s))$

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \ \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 \ m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m \ a \rightarrow m \ a \rightarrow m \ a$

$\text{f } m_1 \ m_2 = \mathbf{do} \ m_1$

$a \leftarrow m_1$

$\text{State } (\lambda s \rightarrow (\dots, s))$

$\text{State } (\lambda s \rightarrow (\dots, s))$

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \ \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 \ m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m \ a \rightarrow m \ a \rightarrow m \ a$

$\text{f } m_1 \ m_2 = \text{do } m_1$

$a \leftarrow m_1$

$\text{State } (\lambda s \rightarrow (\dots, s))$

$\text{State } (\lambda s \rightarrow (\dots, s))$

$(\text{State } (\lambda s \rightarrow (\dots, s))) \gg (\text{State } (\lambda s \rightarrow (\dots, s))) = ?$

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \ \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 \ m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m \ a \rightarrow m \ a \rightarrow m \ a$

$\text{f } m_1 \ m_2 = \text{do } m_1$

$a \leftarrow m_1$

$\text{State } (\lambda s \rightarrow (\dots, s))$

$(\text{State } (\lambda s \rightarrow (\dots, s))) \gg (\text{State } (\lambda s \rightarrow (\dots, s))) = ?$

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \ \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 \ m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m \ a \rightarrow m \ a \rightarrow m \ a$

$\text{f } m_1 \ m_2 = \mathbf{do} \ m_1$

$a \leftarrow m_1$

$\text{State } (\lambda s \rightarrow (\dots, s))$

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \ \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 \ m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m \ a \rightarrow m \ a \rightarrow m \ a$

$\text{f } m_1 \ m_2 = \text{do } m_1$

$a \leftarrow \text{State } (\lambda s \rightarrow (\dots, s))$

$\text{State } (\lambda s \rightarrow (\dots, s))$

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$

$\text{f } m_1 m_2 = \text{do } m_1$

$a \leftarrow \text{State } (\lambda s \rightarrow (\dots, s))$

$\text{State } (\lambda s \rightarrow (\dots, s))$

$(\text{State } (\lambda s \rightarrow (\dots, s))) \gg= (\lambda a \rightarrow \text{State } (\lambda s \rightarrow (\dots, s))) = ?$

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \ \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 \ m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m \ a \rightarrow m \ a \rightarrow m \ a$

$\text{f } m_1 \ m_2 = \text{do } m_1$

$\text{State } (\lambda s \rightarrow (\dots, s))$

$(\text{State } (\lambda s \rightarrow (\dots, s))) \gg= (\lambda a \rightarrow \text{State } (\lambda s \rightarrow (\dots, s))) = ?$

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \ \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 \ m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m \ a \rightarrow m \ a \rightarrow m \ a$

$\text{f } m_1 \ m_2 = \mathbf{do} \ m_1$

$\text{State } (\lambda s \rightarrow (\dots, s))$

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \ \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\mathbf{f} \ m_1 \ m_2) = \text{id}$?

$\mathbf{f} :: \text{Monad } m \Rightarrow m \ a \rightarrow m \ a \rightarrow m \ a$

$\mathbf{f} \ m_1 \ m_2 = \mathbf{do} \ \text{State } (\lambda s \rightarrow (\dots, s))$

$\text{State } (\lambda s \rightarrow (\dots, s))$

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \ \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 \ m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m \ a \rightarrow m \ a \rightarrow m \ a$

$\text{f } m_1 \ m_2 = \text{do } \text{State } (\lambda s \rightarrow (\dots, s))$
 $\text{State } (\lambda s \rightarrow (\dots, s))$

$(\text{State } (\lambda s \rightarrow (\dots, s))) \gg (\text{State } (\lambda s \rightarrow (\dots, s))) = ?$

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \ \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 \ m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m \ a \rightarrow m \ a \rightarrow m \ a$
 $\text{f } m_1 \ m_2 = \text{do State } (\lambda s \rightarrow (\dots, s))$

$(\text{State } (\lambda s \rightarrow (\dots, s))) \gg (\text{State } (\lambda s \rightarrow (\dots, s))) = ?$

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \ \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\mathbf{f} \ m_1 \ m_2) = \text{id}$?

$\mathbf{f} :: \text{Monad } m \Rightarrow m \ a \rightarrow m \ a \rightarrow m \ a$
 $\mathbf{f} \ m_1 \ m_2 = \mathbf{do} \ \text{State } (\lambda s \rightarrow (\dots, s))$

Yes!

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \ \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 \ m_2) = \text{id}$?

```
f :: Monad m => m a -> m a -> m a
f m1 m2 = do State (\s -> (... , s))
```

Yes!

What about other invariants, other monads, ...?

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$

```
 $\text{f } m_1 m_2 = \text{do } m_1$   
     $a \leftarrow m_1$   
     $m_2$   
     $b \leftarrow m_1$   
     $c \leftarrow m_2$   
     $\text{return } b$ 
```

What about other invariants, other monads, ...?

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$

$\text{f } m_1 m_2 = \text{do } m_1$

$a \leftarrow m_1$

m_2

$b \leftarrow m_1$

$c \leftarrow m_2$

$\text{return } b$

“induction over normal form”

What about other invariants, other monads, ...?

Propagating Invariants

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but $\text{execState } m_i = \text{id}$.

Can we show that $\text{execState } (\text{f } m_1 m_2) = \text{id}$?

$\text{f} :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$

$\text{f } m_1 m_2 = \text{do } m_1$

$a \leftarrow m_1$

m_2

$b \leftarrow m_1$

$c \leftarrow m_2$

$\text{return } b$

“induction over normal form”

[Prehofer '99]

What about other invariants, other monads, ...?

Consider a More Specific Type

Instead of

$$f :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$$

now

$$f :: \text{Monad } m \Rightarrow m \text{ Int} \rightarrow m \text{ Int} \rightarrow m \text{ Int}$$

Consider a More Specific Type

Instead of

$$f :: \text{Monad } m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$$

now

$$f :: \text{Monad } m \Rightarrow m\ \text{Int} \rightarrow m\ \text{Int} \rightarrow m\ \text{Int}$$

Then more possible behaviours of `f` are possible:

$$f :: \text{Monad } m \Rightarrow m\ \text{Int} \rightarrow m\ \text{Int} \rightarrow m\ \text{Int}$$
$$f\ m_1\ m_2 = \mathbf{do}\ m_1$$
$$a \leftarrow m_1$$
$$m_2$$
$$b \leftarrow m_1$$
$$c \leftarrow m_2$$
$$\mathbf{return}\ b$$

Consider a More Specific Type

Instead of

```
f :: Monad m => m a -> m a -> m a
```

now

```
f :: Monad m => m Int -> m Int -> m Int
```

Then more possible behaviours of `f` are possible:

```
f :: Monad m => m Int -> m Int -> m Int
```

```
f m1 m2 = do m1
```

```
  a ← m1
```

```
  m2
```

```
  b ← m1
```

```
  if b > 0 then return (a + b)
```

```
    else do c ← m2
```

```
      return b
```

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \ \tau$, but `execState` $m_j = \text{id}$.

`f` :: Monad $m \Rightarrow m \ a \rightarrow m \ a \rightarrow m \ a$

```
f m1 m2 = do m1
           a ← m1
           m2
           b ← m1
           c ← m2
           return b
```

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but `execState` $m_j = \text{id}$.

An m has this property iff it is an h -image for

$h :: \text{Reader } \sigma a \rightarrow \text{State } \sigma a$
 $h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g\ s, s))$

`f` :: Monad $m \Rightarrow m\ a \rightarrow m\ a \rightarrow m\ a$

```
f m1 m2 = do m1
           a <- m1
           m2
           b <- m1
           c <- m2
           return b
```

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but `execState` $m_i = \text{id}$.

An m has this property iff it is an h -image for

$h :: \text{Reader } \sigma a \rightarrow \text{State } \sigma a$
 $h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g s, s))$

Then:

`f` :: Monad $m \Rightarrow m a \rightarrow m a \rightarrow m a$

<code>f</code> ($h m'_1$) ($h m'_2$) = <code>do</code>	<code>f</code> m'_1 m'_2 = <code>do</code>
<code>h m'_1</code>	<code>m'_1</code>
<code>a</code> \leftarrow <code>h m'_1</code>	<code>a</code> \leftarrow <code>m'_1</code>
<code>h m'_2</code>	<code>m'_2</code>
<code>b</code> \leftarrow <code>h m'_1</code>	<code>b</code> \leftarrow <code>m'_1</code>
<code>c</code> \leftarrow <code>h m'_2</code>	<code>c</code> \leftarrow <code>m'_2</code>
<code>return</code> <code>b</code>	<code>return</code> <code>b</code>

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but `execState` $m_i = \text{id}$.

An m has this property iff it is an h -image for

$h :: \text{Reader } \sigma a \rightarrow \text{State } \sigma a$
 $h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g s, s))$

Then:

`f` :: Monad $m \Rightarrow m a \rightarrow m a \rightarrow m a$

<code>f (h m'_1) (h m'_2) = do</code>	<code>h m'_1</code>	<code>f m'_1 m'_2 = do</code>	<code>m'_1</code>
	<code>a ← h m'_1</code>		<code>a ← m'_1</code>
	<code>h m'_2</code>		<code>m'_2</code>
	<code>b ← h m'_1</code>		<code>b ← m'_1</code>
	<code>c ← h m'_2</code>		<code>c ← m'_2</code>
	<code>return b</code>		<code>return b</code>

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but `execState` $m_i = \text{id}$.

An m has this property iff it is an h -image for

$h :: \text{Reader } \sigma a \rightarrow \text{State } \sigma a$
 $h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g s, s))$

Then:

`f` :: Monad $m \Rightarrow m a \rightarrow m a \rightarrow m a$

<code>f</code> ($h m'_1$) ($h m'_2$) = <code>do</code> $h m'_1$	<code>f</code> $m'_1 m'_2$ = <code>do</code> m'_1
$a \leftarrow h m'_1$	$a \leftarrow m'_1$
$h m'_2$	m'_2
$b \leftarrow h m'_1$	$b \leftarrow m'_1$
$c \leftarrow h m'_2$	$c \leftarrow m'_2$
<code>return</code> b	<code>return</code> b

`return` b = h (`return` b)

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but `execState` $m_i = \text{id}$.

An m has this property iff it is an h -image for

$$h :: \text{Reader } \sigma a \rightarrow \text{State } \sigma a$$
$$h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g s, s))$$

Then:

$$f :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$$

$f (h m'_1) (h m'_2) =$	do	$h m'_1$	$f m'_1 m'_2 =$	do	m'_1
		$a \leftarrow h m'_1$			$a \leftarrow m'_1$
		$h m'_2$			m'_2
		$b \leftarrow h m'_1$			$b \leftarrow m'_1$
		$c \leftarrow h m'_2$			$c \leftarrow m'_2$
		$h (\text{return } b)$			return b

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but `execState` $m_i = \text{id}$.

An m has this property iff it is an h -image for

$h :: \text{Reader } \sigma a \rightarrow \text{State } \sigma a$
 $h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g s, s))$

Then:

`f` :: Monad $m \Rightarrow m a \rightarrow m a \rightarrow m a$

<code>f (h m'_1) (h m'_2) = do</code>	<code>h m'_1</code>	<code>f m'_1 m'_2 = do</code>	<code>m'_1</code>
	<code>a ← h m'_1</code>		<code>a ← m'_1</code>
	<code>h m'_2</code>		<code>m'_2</code>
	<code>b ← h m'_1</code>		<code>b ← m'_1</code>
	<code>c ← h m'_2</code>		<code>c ← m'_2</code>
	<code>h (return b)</code>		<code>return b</code>

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but `execState` $m_i = \text{id}$.

An m has this property iff it is an h -image for

$h :: \text{Reader } \sigma a \rightarrow \text{State } \sigma a$
 $h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g s, s))$

Then:

`f` :: Monad $m \Rightarrow m a \rightarrow m a \rightarrow m a$

<code>f (h m'_1) (h m'_2) = do</code>	<code>h m'_1</code>	<code>f m'_1 m'_2 = do</code>	<code>m'_1</code>
	<code>a ← h m'_1</code>		<code>a ← m'_1</code>
	<code>h m'_2</code>		<code>m'_2</code>
	<code>b ← h m'_1</code>		<code>b ← m'_1</code>
	<code>c ← h m'_2</code>		<code>c ← m'_2</code>
	<code>h (return b)</code>		<code>return b</code>

`(h m'_2) >>= (\lambda c \rightarrow h (return b)) = ?`

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but `execState` $m_i = \text{id}$.

An m has this property iff it is an h -image for

$$\begin{aligned} h &:: \text{Reader } \sigma a \rightarrow \text{State } \sigma a \\ h (\text{Reader } g) &= \text{State } (\lambda s \rightarrow (g s, s)) \end{aligned}$$

Then:

$$f :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$$

$$\begin{array}{ll} f (h m'_1) (h m'_2) = \mathbf{do} & f m'_1 m'_2 = \mathbf{do} \\ & a \leftarrow h m'_1 & a \leftarrow m'_1 \\ & h m'_2 & m'_2 \\ & b \leftarrow h m'_1 & b \leftarrow m'_1 \\ & c \leftarrow h m'_2 & c \leftarrow m'_2 \\ & h (\mathbf{return} b) & \mathbf{return} b \end{array}$$

$$(h m'_2) \gg= (\lambda c \rightarrow h (\mathbf{return} b)) = h (m'_2 \gg= (\lambda c \rightarrow \mathbf{return} b))$$

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but `execState` $m_i = \text{id}$.

An m has this property iff it is an h -image for

$$\begin{aligned} h &:: \text{Reader } \sigma a \rightarrow \text{State } \sigma a \\ h (\text{Reader } g) &= \text{State } (\lambda s \rightarrow (g s, s)) \end{aligned}$$

Then:

$$f :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$$

$$\begin{array}{ll} f (h m'_1) (h m'_2) = \mathbf{do} & f m'_1 m'_2 = \mathbf{do} \\ & a \leftarrow h m'_1 & a \leftarrow m'_1 \\ & h m'_2 & m'_2 \\ & b \leftarrow h m'_1 & b \leftarrow m'_1 \\ & h (\mathbf{do} c \leftarrow m'_2 & c \leftarrow m'_2 \\ & \quad \mathbf{return} b) & \mathbf{return} b \end{array}$$

$$(h m'_2) \gg= (\lambda c \rightarrow h (\mathbf{return} b)) = h (m'_2 \gg= (\lambda c \rightarrow \mathbf{return} b))$$

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but `execState` $m_i = \text{id}$.

An m has this property iff it is an h -image for

$$h :: \text{Reader } \sigma a \rightarrow \text{State } \sigma a$$
$$h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g s, s))$$

Then:

$$f :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$$

$f (h m'_1) (h m'_2) =$	<code>do</code>	$h m'_1$		$f m'_1 m'_2 =$	<code>do</code>	m'_1		
		$a \leftarrow$	$h m'_1$			$a \leftarrow$	m'_1	
		$h m'_2$				m'_2		
		$b \leftarrow$	$h m'_1$			$b \leftarrow$	m'_1	
		h	<code>(do</code>	$c \leftarrow$	m'_2	<code>c</code>	\leftarrow	m'_2
			<code>return</code>	$b)$		<code>return</code>	b	

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but `execState` $m_i = \text{id}$.

An m has this property iff it is an h -image for

$$h :: \text{Reader } \sigma a \rightarrow \text{State } \sigma a$$
$$h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g s, s))$$

Then:

$$f :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$$

$f (h m'_1) (h m'_2) =$	do	$h m'_1$		$f m'_1 m'_2 =$	do	m'_1	
		$a \leftarrow$	$h m'_1$			$a \leftarrow$	m'_1
		$h m'_2$				m'_2	
		$b \leftarrow$	$h m'_1$			$b \leftarrow$	m'_1
		h	(do	$c \leftarrow$		$c \leftarrow$	m'_2
			return	$b)$		return	b

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but `execState` $m_i = \text{id}$.

An m has this property iff it is an h -image for

$$h :: \text{Reader } \sigma a \rightarrow \text{State } \sigma a$$
$$h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g s, s))$$

Then:

$$f :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$$

$f (h m'_1) (h m'_2) =$	do $h m'_1$	$f m'_1 m'_2 =$	do m'_1
	$a \leftarrow h m'_1$		$a \leftarrow m'_1$
	$h m'_2$		m'_2
	$b \leftarrow h m'_1$		$b \leftarrow m'_1$
	$h (\mathbf{do} c \leftarrow m'_2$		$c \leftarrow m'_2$
	$\mathbf{return } b)$		$\mathbf{return } b$

$$(h m) \gg= (h \circ k) = ?$$

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but `execState` $m_i = \text{id}$.

An m has this property iff it is an h -image for

$$\begin{aligned} h &:: \text{Reader } \sigma a \rightarrow \text{State } \sigma a \\ h (\text{Reader } g) &= \text{State } (\lambda s \rightarrow (g s, s)) \end{aligned}$$

Then:

$$f :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$$

$$\begin{array}{ll} f (h m'_1) (h m'_2) = \mathbf{do} & f m'_1 m'_2 = \mathbf{do} \\ & a \leftarrow h m'_1 & a \leftarrow m'_1 \\ & h m'_2 & m'_2 \\ & b \leftarrow h m'_1 & b \leftarrow m'_1 \\ & h (\mathbf{do} c \leftarrow m'_2 & c \leftarrow m'_2 \\ & \quad \mathbf{return} b) & \mathbf{return} b \end{array}$$

$$(h m) \gg= (h \circ k) = h (m \gg= k)$$

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but `execState` $m_i = \text{id}$.

An m has this property iff it is an h -image for

$$\begin{aligned} h &:: \text{Reader } \sigma a \rightarrow \text{State } \sigma a \\ h (\text{Reader } g) &= \text{State } (\lambda s \rightarrow (g s, s)) \end{aligned}$$

Then:

$$f :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$$

$$\begin{array}{ll} f (h m'_1) (h m'_2) = \mathbf{do} & f m'_1 m'_2 = \mathbf{do} \\ & a \leftarrow h m'_1 & a \leftarrow m'_1 \\ & h m'_2 & m'_2 \\ & h (\mathbf{do} & b \leftarrow m'_1 \\ & c \leftarrow m'_2 \\ & \mathbf{return } b) & c \leftarrow m'_2 \\ & & \mathbf{return } b \end{array}$$

$$(h m) \gg= (h \circ k) = h (m \gg= k)$$

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but `execState` $m_i = \text{id}$.

An m has this property iff it is an h -image for

$h :: \text{Reader } \sigma a \rightarrow \text{State } \sigma a$
 $h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g s, s))$

Then:

`f` :: Monad $m \Rightarrow m a \rightarrow m a \rightarrow m a$

<code>f (h m'_1) (h m'_2) = do</code>	<code>h m'_1</code>	<code>f m'_1 m'_2 = do</code>	<code>m'_1</code>
	<code> a ← h m'_1</code>		<code> a ← m'_1</code>
	<code> h m'_2</code>		<code> m'_2</code>
	<code> h (do</code>		<code> b ← m'_1</code>
	<code> b ← m'_1</code>		<code> c ← m'_2</code>
	<code> c ← m'_2</code>		<code> return b</code>
	<code> return b)</code>		

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \ \tau$, but `execState` $m_i = \text{id}$.

An m has this property iff it is an h -image for

$h :: \text{Reader } \sigma \ a \rightarrow \text{State } \sigma \ a$
 $h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g \ s, s))$

Then:

`f` :: Monad $m \Rightarrow m \ a \rightarrow m \ a \rightarrow m \ a$

`f` ($h \ m'_1$) ($h \ m'_2$) = **do** $h \ m'_1$
 $a \leftarrow h \ m'_1$
 h (**do** m'_2
 $b \leftarrow m'_1$
 $c \leftarrow m'_2$
 return b)

`f` $m'_1 \ m'_2$ = **do** m'_1
 $a \leftarrow m'_1$
 m'_2
 $b \leftarrow m'_1$
 $c \leftarrow m'_2$
 return b

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \ \tau$, but `execState` $m_i = \text{id}$.

An m has this property iff it is an h -image for

$h :: \text{Reader } \sigma \ a \rightarrow \text{State } \sigma \ a$
 $h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g \ s, s))$

Then:

`f` :: Monad $m \Rightarrow m \ a \rightarrow m \ a \rightarrow m \ a$

`f` ($h \ m'_1$) ($h \ m'_2$) = `do` $h \ m'_1$
 h (`do` $a \leftarrow m'_1$
 m'_2
 $b \leftarrow m'_1$
 $c \leftarrow m'_2$
`return` b)

`f` $m'_1 \ m'_2$ = `do` m'_1
 $a \leftarrow m'_1$
 m'_2
 $b \leftarrow m'_1$
 $c \leftarrow m'_2$
`return` b

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but `execState` $m_i = \text{id}$.

An m has this property iff it is an h -image for

$$h :: \text{Reader } \sigma a \rightarrow \text{State } \sigma a$$
$$h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g s, s))$$

Then:

$$f :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$$
$$f (h m'_1) (h m'_2) = \text{do } h (\text{do } m'_1$$
$$a \leftarrow m'_1$$
$$m'_2$$
$$b \leftarrow m'_1$$
$$c \leftarrow m'_2$$
$$\text{return } b)$$
$$f m'_1 m'_2 = \text{do } m'_1$$
$$a \leftarrow m'_1$$
$$m'_2$$
$$b \leftarrow m'_1$$
$$c \leftarrow m'_2$$
$$\text{return } b$$

Reasoning via Monad Embedding

Assume $m_1, m_2 :: \text{State } \sigma \tau$, but `execState` $m_i = \text{id}$.

An m has this property iff it is an h -image for

$$h :: \text{Reader } \sigma a \rightarrow \text{State } \sigma a$$
$$h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g s, s))$$

Then:

$$f :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$$

<code>f</code>	<code>(h m'_1) (h m'_2)</code>	<code>= do h (do m'_1</code>	<code>f m'_1 m'_2 = do m'_1</code>
		<code> a ← m'_1</code>	<code> a ← m'_1</code>
		<code> m'_2</code>	<code> m'_2</code>
		<code> b ← m'_1</code>	<code> b ← m'_1</code>
		<code> c ← m'_2</code>	<code> c ← m'_2</code>
		<code> return b)</code>	<code> return b</code>

$$f (h m'_1) (h m'_2) = h (f m'_1 m'_2)$$

A More General Theorem

Let

$$f :: \text{Monad } m \Rightarrow m \text{ Int} \rightarrow m \text{ Int} \rightarrow m \text{ Int}$$

Let

$$h :: \kappa_1 a \rightarrow \kappa_2 a$$

such that

- ▶ κ_1, κ_2 are monads
- ▶ $h \circ \text{return}_{\kappa_1} = \text{return}_{\kappa_2}$
- ▶ for every m and k , $h (m \gg=_{\kappa_1} k) = (h m) \gg=_{\kappa_2} (h \circ k)$

Then for every m_1 and m_2 ,

$$f (h m_1) (h m_2) = h (f m_1 m_2)$$

The same for

$$f :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$$

A More General Theorem

Let

$$f :: \text{Monad } m \Rightarrow m \text{ Int} \rightarrow m \text{ Int} \rightarrow m \text{ Int}$$

Let

$$h :: \kappa_1 a \rightarrow \kappa_2 a$$

such that

- ▶ κ_1, κ_2 are monads
- ▶ $h \circ \text{return}_{\kappa_1} = \text{return}_{\kappa_2}$
- ▶ for every m and k , $h (m \gg=_{\kappa_1} k) = (h m) \gg=_{\kappa_2} (h \circ k)$

Then for every m_1 and m_2 ,

$$f (h m_1) (h m_2) = h (f m_1 m_2)$$

The same for

$$f :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$$

Looking Back at the Concrete Invariant

For

$$h :: \text{Reader } \sigma \ a \rightarrow \text{State } \sigma \ a$$
$$h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g \ s, s))$$

Looking Back at the Concrete Invariant

For

$$h :: \text{Reader } \sigma \ a \rightarrow \text{State } \sigma \ a$$

$$h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g \ s, s))$$

the conditions

▶ $h \circ \text{return}_{\text{Reader } \sigma} = \text{return}_{\text{State } \sigma}$

▶ for every m and k ,

$$h (m \gg=_{\text{Reader } \sigma} k) = (h \ m) \gg=_{\text{State } \sigma} (h \circ k)$$

Looking Back at the Concrete Invariant

For

$$h :: \text{Reader } \sigma \ a \rightarrow \text{State } \sigma \ a$$
$$h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g \ s, s))$$

the conditions

- ▶ $h \circ \text{return}_{\text{Reader } \sigma} = \text{return}_{\text{State } \sigma}$
- ▶ for every m and k ,
 $h (m \gg=_{\text{Reader } \sigma} k) = (h \ m) \gg=_{\text{State } \sigma} (h \circ k)$

imply that

- ▶ for every a , $\text{execState} (\text{return}_{\text{State } \sigma} \ a) = \text{id}$

Looking Back at the Concrete Invariant

For

$$h :: \text{Reader } \sigma \ a \rightarrow \text{State } \sigma \ a$$
$$h (\text{Reader } g) = \text{State } (\lambda s \rightarrow (g \ s, s))$$

the conditions

- ▶ $h \circ \text{return}_{\text{Reader } \sigma} = \text{return}_{\text{State } \sigma}$
- ▶ for every m and k ,
 $h (m \gg=_{\text{Reader } \sigma} k) = (h \ m) \gg=_{\text{State } \sigma} (h \circ k)$

imply that

- ▶ for every a , $\text{execState } (\text{return}_{\text{State } \sigma} \ a) = \text{id}$
- ▶ for every m and k , $\text{execState } (m \gg=_{\text{State } \sigma} k) = \text{id}$,
provided:
 - ▶ $\text{execState } m = \text{id}$
 - ▶ for every a , $\text{execState } (k \ a) = \text{id}$

A More General Theorem

Let

$$f :: \text{Monad } m \Rightarrow m \text{ Int} \rightarrow m \text{ Int} \rightarrow m \text{ Int}$$

Let

$$h :: \kappa_1 a \rightarrow \kappa_2 a$$

such that

- ▶ κ_1, κ_2 are monads
- ▶ $h \circ \text{return}_{\kappa_1} = \text{return}_{\kappa_2}$
- ▶ for every m and k , $h (m \gg=_{\kappa_1} k) = (h m) \gg=_{\kappa_2} (h \circ k)$

Then for every m_1 and m_2 ,

$$f (h m_1) (h m_2) = h (f m_1 m_2)$$

The same for

$$f :: \text{Monad } m \Rightarrow m a \rightarrow m a \rightarrow m a$$

Conceptual Ingredients

- ▶ Exploiting polymorphism
 - ▶ Relational parametricity [Reynolds '83]
 - ▶ Free theorems [Wadler '89]

Conceptual Ingredients

- ▶ Exploiting polymorphism
 - ▶ Relational parametricity [Reynolds '83]
 - ▶ Free theorems [Wadler '89]
- ▶ Extension to type classes:
 - ▶ Folklore
 - ▶ via dictionary translation [Wadler & Blott '89]

Conceptual Ingredients

- ▶ Exploiting polymorphism
 - ▶ Relational parametricity [Reynolds '83]
 - ▶ Free theorems [Wadler '89]
- ▶ Extension to type classes:
 - ▶ Folklore
 - ▶ via dictionary translation [Wadler & Blott '89]
- ▶ Extension to type constructors:
 - ▶ Folklore?
 - ▶ a recent formal account [Vytiniotis & Weirich '10]

Conceptual Ingredients

- ▶ Exploiting polymorphism
 - ▶ Relational parametricity [Reynolds '83]
 - ▶ Free theorems [Wadler '89]
- ▶ Extension to type classes:
 - ▶ Folklore
 - ▶ via dictionary translation [Wadler & Blott '89]
- ▶ Extension to type constructors:
 - ▶ Folklore?
 - ▶ a recent formal account [Vytiniotis & Weirich '10]
- ▶ Monad morphisms:
 - ▶ Representation independence for effects [Filinski & Støvring '07, Filinski '07]

Example Uses

- ▶ Invariant propagation, e.g.:
 - ▶ Purity
 - ▶ Independence criteria for stateful computations
 - ▶ Restrictions on IO

Example Uses

- ▶ Invariant propagation, e.g.:
 - ▶ Purity
 - ▶ Independence criteria for stateful computations
 - ▶ Restrictions on IO
- ▶ Safe value extraction, e.g.:
 - ▶ Discard logging
 - ▶ Pick from a nondeterministic manifold

Example Uses

- ▶ Invariant propagation, e.g.:
 - ▶ Purity
 - ▶ Independence criteria for stateful computations
 - ▶ Restrictions on IO
- ▶ Safe value extraction, e.g.:
 - ▶ Discard logging
 - ▶ Pick from a nondeterministic manifold
- ▶ Effect abstraction, e.g.
 - ▶ From exceptions to partiality

Example Uses

- ▶ Invariant propagation, e.g.:
 - ▶ Purity
 - ▶ Independence criteria for stateful computations
 - ▶ Restrictions on IO
- ▶ Safe value extraction, e.g.:
 - ▶ Discard logging
 - ▶ Pick from a nondeterministic manifold
- ▶ Effect abstraction, e.g.
 - ▶ From exceptions to partiality
- ▶ Proper generalisations of standard free theorems

Example Uses

- ▶ Invariant propagation, e.g.:
 - ▶ Purity
 - ▶ Independence criteria for stateful computations
 - ▶ Restrictions on IO
- ▶ Safe value extraction, e.g.:
 - ▶ Discard logging
 - ▶ Pick from a nondeterministic manifold
- ▶ Effect abstraction, e.g.
 - ▶ From exceptions to partiality
- ▶ Proper generalisations of standard free theorems
- ▶ Transparent introduction of data type improvements [V. '08b]

Example Uses

- ▶ Invariant propagation, e.g.:
 - ▶ Purity
 - ▶ Independence criteria for stateful computations
 - ▶ Restrictions on IO
- ▶ Safe value extraction, e.g.:
 - ▶ Discard logging
 - ▶ Pick from a nondeterministic manifold
- ▶ Effect abstraction, e.g.
 - ▶ From exceptions to partiality
- ▶ Proper generalisations of standard free theorems
- ▶ Transparent introduction of data type improvements [V. '08b]
- ▶ Reasoning about “harmless advice” [Oliveira et al. '10]

EffectiveAdvice: Disciplined Advice with Explicit Effects

Advice/AOP:

- ▶ Separation of cross-cutting concerns in software

EffectiveAdvice: Disciplined Advice with Explicit Effects

Advice/AOP:

- ▶ Separation of cross-cutting concerns in software
- ▶ “Weaving”:



EffectiveAdvice: Disciplined Advice with Explicit Effects

Advice/AOP:

- ▶ Separation of cross-cutting concerns in software
- ▶ “Weaving”:



- ▶ Modular analysis/reasoning?

EffectiveAdvice: Disciplined Advice with Explicit Effects

Advice/AOP:

- ▶ Separation of cross-cutting concerns in software
- ▶ “Weaving”:



- ▶ Modular analysis/reasoning?

EffectiveAdvice [Oliveira et al. '10]:

- ▶ semantic model, à la Open Modules [Aldrich '05]

EffectiveAdvice: Disciplined Advice with Explicit Effects

Advice/AOP:

- ▶ Separation of cross-cutting concerns in software
- ▶ “Weaving”:



- ▶ Modular analysis/reasoning?

EffectiveAdvice [Oliveira et al. '10]:

- ▶ semantic model, à la Open Modules [Aldrich '05]
- ▶ allows modular reasoning ...

EffectiveAdvice: Disciplined Advice with Explicit Effects

Advice/AOP:

- ▶ Separation of cross-cutting concerns in software
- ▶ “Weaving”:



- ▶ Modular analysis/reasoning?

EffectiveAdvice [Oliveira et al. '10]:

- ▶ semantic model, à la Open Modules [Aldrich '05]
- ▶ allows modular reasoning ...
- ▶ in the presence of side effects!

EffectiveAdvice: Disciplined Advice with Explicit Effects

Specifics from AOP perspective:

- ▶ components explicitly specify their “entry points for advice”
 - types, open recursion

EffectiveAdvice: Disciplined Advice with Explicit Effects

Specifics from AOP perspective:

- ▶ components explicitly specify their “entry points for advice”
 - types, open recursion
- ▶ advice composition is explicit
 - combinator library

EffectiveAdvice: Disciplined Advice with Explicit Effects

Specifics from AOP perspective:

- ▶ components explicitly specify their “entry points for advice”
 - types, open recursion
- ▶ advice composition is explicit
 - combinator library
- ▶ components state the effects they are using
 - monads, transformers, specialized classes

EffectiveAdvice: Disciplined Advice with Explicit Effects

Specifics from AOP perspective:

- ▶ components explicitly specify their “entry points for advice”
 - types, open recursion
- ▶ advice composition is explicit
 - combinator library
- ▶ components state the effects they are using
 - monads, transformers, specialized classes

Benefits:

- ▶ equational reasoning

EffectiveAdvice: Disciplined Advice with Explicit Effects

Specifics from AOP perspective:

- ▶ components explicitly specify their “entry points for advice”
 - types, open recursion
- ▶ advice composition is explicit
 - combinator library
- ▶ components state the effects they are using
 - monads, transformers, specialized classes

Benefits:

- ▶ equational reasoning
- ▶ type shapes (higher-rank) capture interference patterns

EffectiveAdvice: Disciplined Advice with Explicit Effects

Specifics from AOP perspective:

- ▶ components explicitly specify their “entry points for advice”
 - types, open recursion
- ▶ advice composition is explicit
 - combinator library
- ▶ components state the effects they are using
 - monads, transformers, specialized classes

Benefits:

- ▶ equational reasoning
- ▶ type shapes (higher-rank) capture interference patterns
- ▶ correctness proofs about non-interference

EffectiveAdvice: Disciplined Advice with Explicit Effects

Theorem 2 (Harmless Observation Advice) [Oliveira et al. '10]:

Consider any base program and any advice with the types:

$\text{bse} :: \forall t. (\text{MonadTrans } t, \dots) \Rightarrow \text{Open } (\dots)$

$\text{adv} :: \forall m. \text{MGet } \sigma \ m \Rightarrow \text{Augment } \dots$

If a function $\text{proj} :: \forall m \ a. \text{Monad } m \Rightarrow \tau \ m \ a \rightarrow m \ a$ exists that satisfies \dots , then advice adv is harmless with respect to bse :

$\text{proj} \circ (\text{weave } (\text{adv} \odot \text{bse})) = \text{runIdT} \circ (\text{weave } \text{bse})$

References I



J. Aldrich.

Open Modules: Modular reasoning about advice.

In *European Conference on Object-Oriented Programming, Proceedings*, volume 3586 of *LNCS*, pages 144–168.

Springer-Verlag, 2005.



J.-P. Bernardy, P. Jansson, and K. Claessen.

Testing polymorphic properties.

In *European Symposium on Programming, Proceedings*, volume 6012 of *LNCS*, pages 125–144. Springer-Verlag, 2010






A. Filinski.

On the relations between monadic semantics.

Theoretical Computer Science, 375(1–3):41–75, 2007.

References II

-  A. Filinski and K. Støvring.
Inductive reasoning about effectful data types.
In *International Conference on Functional Programming, Proceedings*, pages 97–110. ACM Press, 2007.
-  A. Gill, J. Launchbury, and S.L. Peyton Jones.
A short cut to deforestation.
In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 223–232. ACM Press, 1993.
-  G. Hutton and D. Fulger.
Reasoning about effects: Seeing the wood through the trees.
In *Trends in Functional Programming, Draft Proceedings*, 2008.

References III



E. Moggi.

Notions of computation and monads.

Information and Computation, 93(1):55–92, 1991.



B.C.d.S. Oliveira, T. Schrijvers, and W.R. Cook.

EffectiveAdvice: Disciplined advice with explicit effects.

In *Aspect-Oriented Software Development, Proceedings*, pages 109–120. ACM Press, 2010.



S.L. Peyton Jones.

Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell.

In *Engineering Theories of Software Construction, Marktoberdorf Summer School 2000, Proceedings*, pages 47–96. IOS Press, 2001.

References IV



S.L. Peyton Jones and P. Wadler.

Imperative functional programming.

In *Principles of Programming Languages, Proceedings*, pages 71–84. ACM Press, 1993.



C. Prehofer.

Flexible construction of software components: A feature oriented approach.

Habilitation Thesis, Technische Universität München, 1999.



J.C. Reynolds.

Types, abstraction and parametric polymorphism.

In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.

References V



W. Swierstra and T. Altenkirch.

Beauty in the beast — A functional semantics for the awkward squad.

In *Haskell Workshop, Proceedings*, pages 25–36. ACM Press, 2007.



J. Voigtländer.

Much ado about two: A pearl on parallel prefix computation.

In *Principles of Programming Languages, Proceedings*, pages 29–35. ACM Press, 2008a.



J. Voigtländer.

Asymptotic improvement of computations over free monads.

In *Mathematics of Program Construction, Proceedings*, volume 5133 of *LNCS*, pages 388–403. Springer-Verlag, 2008b.

References VI



J. Voigtländer.

Bidirectionalization for free!

In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM Press, 2009a.



J. Voigtländer.

Free theorems involving type constructor classes.

In *International Conference on Functional Programming, Proceedings*, pages 173–184. ACM Press, 2009b.



D. Vytiniotis and S. Weirich.

Parametricity, type equality, and higher-order polymorphism.

Journal of Functional Programming, 20(2):175–210, 2010.

References VII



P. Wadler.

Theorems for free!

In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.



P. Wadler.

The essence of functional programming (Invited talk).

In *Principles of Programming Languages, Proceedings*, pages 1–14. ACM Press, 1992.



P. Wadler and S. Blott.

How to make ad-hoc polymorphism less ad hoc.

In *Principles of Programming Languages, Proceedings*, pages 60–76. ACM Press, 1989.