

Seeing the wood through the trees

A DIY guide to reasoning about effects

Graham Hutton and Diana Fulger

University of Nottingham

December 16, 2007

Background

Monads

Background

Monads

- make writing programmes easier

Background

Monads

- make writing programmes easier
- should make reasoning about them easier too?

Background

Monads

- make writing programmes easier
- should make reasoning about them easier too?

But

Background

Monads

- make writing programmes easier
- should make reasoning about them easier too?

But existing techniques have not gone mainstream

Background

Monads

- make writing programmes easier
- should make reasoning about them easier too?

But existing techniques have not gone mainstream

Let's try a simple example!

Reasoning on trees

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

Reasoning on trees

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

label      :: Tree a → Int → (Tree Int, Int)
label (Leaf x)  n = (Leaf n, n+1)
label (Node l r) n = (Node l' r', n'')
                where (l', n') = label l n
                      (r', n'') = label r n'
```

Reasoning on trees

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

```
label :: Tree a → Int → (Tree Int, Int)
```

```
label (Leaf x) n = (Leaf n, n+1)
```

```
label (Node l r) n = (Node l' r', n")
```

```
    where (l', n') = label l n
```

```
          (r', n") = label r n'
```

Prove label correct?

Reasoning on trees

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

label      :: Tree a → Int → (Tree Int, Int)
label (Leaf x)  n = (Leaf n, n+1)
label (Node l r) n = (Node l' r', n'')
                  where (l', n') = label l n
                        (r', n'') = label r n'
```

Prove label correct? **equationally!**

Define “correct”?

Specification

- preserves shape

Define “correct”?

Specification

- preserves shape
- **distinct labels**

Simple correctness

```
labels          :: Tree a → [a]
labels (Leaf x)  = [x]
labels (Node l r) = labels l ++ labels r
```

Simple correctness

```
labels          :: Tree a → [a]
labels (Leaf x) = [x]
labels (Node l r) = labels l ++ labels r
```

Theorem

(Correctness of label) For all finite trees t:

$$\text{nodups (labels (fst (label t 0)))}$$

Simple correctness

```
labels          :: Tree a → [a]
labels (Leaf x) = [x]
labels (Node l r) = labels l ++ labels r
```

Theorem

(Correctness of label) For all finite trees t:

$$\text{nodups (labels (fst (label t 0)))}$$

```
nodups          :: Eq a ⇒ [a] → Bool
nodups []       = True
nodups (x:xs)   = not(elem x xs) && nodups xs
```

Proof steps

- identify generated labels

Proof steps

- identify generated labels

Lemma

(Behaviour of label) For every finite tree t and initial integer label n ,

$$\text{label } t \ n = (t', n') \Rightarrow n < n' \wedge \text{labels } t' = [n .. n' - 1]$$

Proof steps

- identify generated labels

Lemma

(Behaviour of label) For every finite tree t and initial integer label n ,

$$\text{label } t \ n = (t', n') \Rightarrow n < n' \wedge \text{labels } t' = [n .. n' - 1]$$

- labels are unique

Proof steps

- identify generated labels

Lemma

(Behaviour of label) For every finite tree t and initial integer label n ,

$$\text{label } t \ n = (t', n') \Rightarrow n < n' \wedge \text{labels } t' = [n .. n' - 1]$$

- labels are unique

Lemma

(Haskell) Intervals have no duplicates: for all integers $a \leq b$:

$$\text{nodups } [a .. b]$$

State monad

```
label :: Tree a → Int → (Tree Int, Int)
```

State monad

```
label :: Tree a → Int → (Tree Int, Int)
```

```
type ST s a = s → (a,s)
```

State monad

```
label :: Tree a → Int → (Tree Int, Int)
```

```
type ST s a = s → (a,s)
```

```
instance Monad (ST s) where
```

```
  -- return :: a → ST s a
```

```
  return v = λs → (v,s)
```

```
  -- (⋈>=>) :: ST s a → (a → ST s b) → ST s b
```

```
  st >>= f = λs → let (v,s') = st s in (f v) s'
```

Monadic labelling

```
label          :: Tree a → Int → (Tree Int, Int)
label (Leaf x) n = (Leaf n, n+1)
label (Node l r) n = (Node l' r', n'')
                    where (l', n') = label l n
                          (r', n'') = label r n'
```

Monadic labelling

```
label :: Tree a → Int → (Tree Int, Int)
label (Leaf x) n = (Leaf n, n+1)
label (Node l r) n = (Node l' r', n'')
                    where (l', n') = label l n
                          (r', n'') = label r n'
```

```
label :: Tree a → ST Int (Tree Int)
label (Leaf x) = do n ← fresh
                  return (Leaf n)
label (Node l r) = do l' ← label l
                      r' ← label r
                      return (Node l' r')
```

Monadic labelling

```
label :: Tree a → Int → (Tree Int, Int)
label (Leaf x) n = (Leaf n, n+1)
label (Node l r) n = (Node l' r', n'')
                    where (l', n') = label l n
                          (r', n'') = label r n'
```

```
label :: Tree a → ST Int (Tree Int)
label (Leaf x) = do n ← fresh
                  return (Leaf n)
label (Node l r) = do l' ← label l
                     r' ← label r
                     return (Node l' r')
```

```
fresh :: ST Int Int
fresh = λn → (n, n+1)
```

Monadic labelling

```
label          :: Tree a → ST Int (Tree Int)
label (Leaf x) = do n ← fresh
                 return (Leaf n)
label (Node l r) = do l' ← label l
                      r' ← label r
                      return (Node l' r')
```

Monadic labelling

```
label          :: Tree a → ST Int (Tree Int)
label (Leaf x) = do n ← fresh
                 return (Leaf n)
label (Node l r) = do l' ← label l
                      r' ← label r
                      return (Node l' r')
```

- single-assignment variables

Monadic labelling

```
label          :: Tree a → ST Int (Tree Int)
label (Leaf x) = do n ← fresh
                 return (Leaf n)
label (Node l r) = do l' ← label l
                      r' ← label r
                      return (Node l' r')
```

- single-assignment variables
- refreshing fresh

Monadic labelling

```
label          :: Tree a → ST Int (Tree Int)
label (Leaf x) = do n ← fresh
                 return (Leaf n)
label (Node l r) = do l' ← label l
                      r' ← label r
                      return (Node l' r')
```

- single-assignment variables
- refreshing fresh
- reasoning any easier?

What we would like to see

Leaf x \rightarrow Leaf fresh

Node l r \rightarrow Node (label l) (label r)

Applicative functors

Effectful generalisation of application

Applicative functors

Effectful generalisation of application

```
class Functor f => Applicative f where
  pure :: a -> f a
  (*) :: f (a -> b) -> f a -> f b
```

Applicative functors

Effectful generalisation of application

```
class Functor f => Applicative f where
  pure :: a -> f a
  (*) :: f (a -> b) -> f a -> f b
```

Monad	→	Applicative
return	→	pure
ap	→	(*)

Applicative functors

```
class Functor f => Applicative f where  
  pure :: a -> f a  
  (*):: f (a -> b) -> f a -> f b
```

Applicative functors

class Functor f \Rightarrow Applicative f where

pure :: a \rightarrow f a

(\otimes) :: f (a \rightarrow b) \rightarrow f a \rightarrow f b

pure id \otimes v = v

identity

pure (o) \otimes u \otimes v \otimes w = u \otimes (v \otimes w)

composition

pure f \otimes pure x = pure (f x)

homomorphism

u \otimes pure y = pure (λ f \rightarrow f y) \otimes u

interchange

Applicative functors

class Functor f \Rightarrow Applicative f where

pure :: a \rightarrow f a

(\otimes) :: f (a \rightarrow b) \rightarrow f a \rightarrow f b

pure id \otimes v = v

identity

pure (o) \otimes u \otimes v \otimes w = u \otimes (v \otimes w)

composition

pure f \otimes pure x = pure (f x)

homomorphism

u \otimes pure y = pure (λ f \rightarrow f y) \otimes u

interchange

pure f \otimes x1 \otimes ... \otimes xn = \llbracket f x1 x2 ... xn \rrbracket

State applicative functor

```
instance Applicative (ST s) where
  -- pure    :: a → ST s a
  pure v     = λs → (v,s)

  --(⊛)     :: ST s (a → b) → ST s a → ST s b
  mf ⊛ mx = λs → let (f, s') = mf s
                    (x, s'') = mx s'
                    in (f x, s'')
```

Applicative labelling

```
label          :: Tree a → ST Int (Tree Int)
```

Applicative labelling

label `:: Tree a → ST Int (Tree Int)`

monadic expression

Applicative labelling

```
label          :: Tree a → ST Int (Tree Int)
```

monadic expression

```
label (Leaf x)  = do n ← fresh  
                  return (Leaf n)  
label (Node l r) = do l' ← label l  
                      r' ← label r  
                      return (Node l' r')
```

Applicative labelling

```
label          :: Tree a → ST Int (Tree Int)
```

monadic expression

```
label (Leaf x)  = do n ← fresh  
                  return (Leaf n)  
label (Node l r) = do l' ← label l  
                      r' ← label r  
                      return (Node l' r')
```

applicative expression

Applicative labelling

```
label :: Tree a → ST Int (Tree Int)
```

monadic expression

```
label (Leaf x) = do n ← fresh
                  return (Leaf n)
label (Node l r) = do l' ← label l
                      r' ← label r
                      return (Node l' r')
```

applicative expression

```
label (Leaf _) = [ Leaf fresh ]
label (Node l r) = [ Node (label l) (label r) ]
```

Applicative approach

Theorem

(Correctness of label) For all finite trees t :

$$\text{nodups} (\text{append} \circ (\text{labels} \times \text{from}) \circ \text{label } t)$$

Applicative approach

Theorem

(Correctness of label) For all finite trees t :

$$\text{nodups} (\text{append} \circ (\text{labels} \times \text{from}) \circ \text{label } t)$$

Lemma

(Behaviour of label) For all finite trees t :

$$\text{append} \circ (\text{labels} \times \text{from}) \circ \text{label } t = \text{from}$$

Applicative labelling

```
label          :: Tree a → ST Int (Tree Int)
label (Leaf _) = [ Leaf fresh ]
label (Node l r) = [ Node (label l) (label r) ]
```

Applicative labelling

```
label          :: Tree a → ST Int (Tree Int)
label (Leaf _) = [ Leaf fresh ]
label (Node l r) = [ Node (label l) (label r) ]
```

- state threading

Applicative labelling

```
label          :: Tree a → ST Int (Tree Int)
label (Leaf _) = [ Leaf fresh ]
label (Node l r) = [ Node (label l) (label r) ]
```

- state threading
- fresh label generation

Applicative labelling

```
label          :: Tree a → ST Int (Tree Int)
label (Leaf _) = [ Leaf fresh ]
label (Node l r) = [ Node (label l) (label r) ]
```

- state threading
- fresh label generation

```
label'         :: Tree a → ST [b] (Tree b)
label' (Leaf _) = [ Leaf fetch ]
label' (Node l r) = [ Node (label l) (label r) ]
```

Applicative labelling

```
label          :: Tree a → ST Int (Tree Int)
label (Leaf _) = [ Leaf fresh ]
label (Node l r) = [ Node (label l) (label r) ]
```

- state threading
- fresh label generation

```
label'         :: Tree a → ST [b] (Tree b)
label' (Leaf _) = [ Leaf fetch ]
label' (Node l r) = [ Node (label l) (label r) ]
```

```
fetch :: [b] → (b, [b])
fetch (x:xs) = (x, xs)
```

Factorisation

Theorem

(Factorising label) For all finite trees t :

$$(\text{id} \times \text{from}) \circ \text{label } t = (\text{label}' t) \circ \text{from}$$

Factorisation

Theorem

(Factorising label) For all finite trees t :

$$(\text{id} \times \text{from}) \circ \text{label } t = (\text{label}' t) \circ \text{from}$$

Proof by induction, using equational reasoning

Base case

$$(\text{id} \times \text{from}) \circ \text{label} (\text{Leaf } _)$$

Base case

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Leaf } _) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Leaf fresh} \rrbracket \end{aligned}$$

Base case

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Leaf } _) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Leaf fresh} \rrbracket \\ = & (\text{id} \times \text{from}) \circ (\text{Leaf} \times \text{id}) \circ \text{fresh} \end{aligned}$$

Base case

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Leaf } _) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Leaf fresh} \rrbracket \\ = & (\text{id} \times \text{from}) \circ (\text{Leaf} \times \text{id}) \circ \text{fresh} \\ = & (\text{Leaf} \times \text{id}) \circ (\text{id} \times \text{from}) \circ \text{fresh} \end{aligned}$$

Base case

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Leaf } _) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Leaf fresh} \rrbracket \\ = & (\text{id} \times \text{from}) \circ (\text{Leaf} \times \text{id}) \circ \text{fresh} \\ = & (\text{Leaf} \times \text{id}) \circ (\text{id} \times \text{from}) \circ \text{fresh} \\ = & (\text{Leaf} \times \text{id}) \circ \text{fetch} \circ \text{from} \end{aligned}$$

Base case

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Leaf } _) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Leaf fresh} \rrbracket \\ = & (\text{id} \times \text{from}) \circ (\text{Leaf} \times \text{id}) \circ \text{fresh} \\ = & (\text{Leaf} \times \text{id}) \circ (\text{id} \times \text{from}) \circ \text{fresh} \\ = & (\text{Leaf} \times \text{id}) \circ \text{fetch} \circ \text{from} \\ = & \llbracket \text{Leaf fetch} \rrbracket \circ \text{from} \end{aligned}$$

Base case

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Leaf } _) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Leaf fresh} \rrbracket \\ = & (\text{id} \times \text{from}) \circ (\text{Leaf} \times \text{id}) \circ \text{fresh} \\ = & (\text{Leaf} \times \text{id}) \circ (\text{id} \times \text{from}) \circ \text{fresh} \\ = & (\text{Leaf} \times \text{id}) \circ \text{fetch} \circ \text{from} \\ = & \llbracket \text{Leaf fetch} \rrbracket \circ \text{from} \\ = & \text{label}' (\text{Leaf } _) \circ \text{from} \end{aligned}$$

Base case

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Leaf } _) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Leaf fresh} \rrbracket \\ = & (\text{id} \times \text{from}) \circ (\text{Leaf} \times \text{id}) \circ \text{fresh} \\ = & (\text{Leaf} \times \text{id}) \circ (\text{id} \times \text{from}) \circ \text{fresh} \\ = & (\text{Leaf} \times \text{id}) \circ \text{fetch} \circ \text{from} \\ = & \llbracket \text{Leaf fetch} \rrbracket \circ \text{from} \\ = & \text{label}' (\text{Leaf } _) \circ \text{from} \end{aligned}$$

Did we just expand ST out

Base case

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Leaf } _) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Leaf fresh} \rrbracket \\ = & (\text{id} \times \text{from}) \circ (\text{Leaf} \times \text{id}) \circ \text{fresh} \\ = & (\text{Leaf} \times \text{id}) \circ (\text{id} \times \text{from}) \circ \text{fresh} \\ = & (\text{Leaf} \times \text{id}) \circ \text{fetch} \circ \text{from} \\ = & \llbracket \text{Leaf fetch} \rrbracket \circ \text{from} \\ = & \text{label}' (\text{Leaf } _) \circ \text{from} \end{aligned}$$

Did we just expand ST out (only to rebuild back again) ??

What ST does...

Lemma

(Bracket notation) For all $f : a \rightarrow b$ and $x : \text{ST } s \ a$:

$$\llbracket f \ x \rrbracket = (f \times \text{id}) \circ x$$

Base case revisited

$$(\text{id} \times \text{from}) \circ \text{label} (\text{Leaf } _)$$

Base case revisited

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Leaf } _) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Leaf fresh} \rrbracket \end{aligned}$$

Base case revisited

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Leaf } _) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Leaf fresh} \rrbracket \\ = & \llbracket \text{Leaf } ((\text{id} \times \text{from}) \circ \text{fresh}) \rrbracket \end{aligned}$$

Base case revisited

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Leaf } _) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Leaf fresh} \rrbracket \\ = & \llbracket \text{Leaf } ((\text{id} \times \text{from}) \circ \text{fresh}) \rrbracket \\ = & \llbracket \text{Leaf } (\text{fetch} \circ \text{from}) \rrbracket \end{aligned}$$

Base case revisited

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Leaf } _) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Leaf fresh} \rrbracket \\ = & \llbracket \text{Leaf } ((\text{id} \times \text{from}) \circ \text{fresh}) \rrbracket \\ = & \llbracket \text{Leaf } (\text{fetch} \circ \text{from}) \rrbracket \\ = & \llbracket \text{Leaf } \text{fetch} \rrbracket \circ \text{from} \end{aligned}$$

Base case revisited

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Leaf } _) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Leaf fresh} \rrbracket \\ = & \llbracket \text{Leaf } ((\text{id} \times \text{from}) \circ \text{fresh}) \rrbracket \\ = & \llbracket \text{Leaf } (\text{fetch} \circ \text{from}) \rrbracket \\ = & \llbracket \text{Leaf fetch} \rrbracket \circ \text{from} \\ = & \text{label}' (\text{Leaf } _) \circ \text{from} \end{aligned}$$

State fusions

Lemma

(Input state fusion) For all $f : a \rightarrow b$, $x : \text{ST } s' a$ and $g : s \rightarrow s'$:

$$\llbracket f \ x \rrbracket \circ g = \llbracket f \ (x \circ g) \rrbracket$$

State fusions

Lemma

(Input state fusion) For all $f : a \rightarrow b$, $x : \text{ST } s' \ a$ and $g : s \rightarrow s'$:

$$\llbracket f \ x \rrbracket \circ g = \llbracket f \ (x \circ g) \rrbracket$$

Typing issue $\llbracket f \ (x \circ g) \rrbracket :: s \rightarrow (s', a)$,

State fusions

Lemma

(Input state fusion) For all $f : a \rightarrow b$, $x : \text{ST } s' \ a$ and $g : s \rightarrow s'$:

$$\llbracket f \ x \rrbracket \circ g = \llbracket f \ (x \circ g) \rrbracket$$

Typing issue $\llbracket f \ (x \circ g) \rrbracket :: s \rightarrow (s', a)$, **not** a valid ST applicative expression!

State fusions

Lemma

(Input state fusion) For all $f : a \rightarrow b$, $x : \text{ST } s' \ a$ and $g : s \rightarrow s'$:

$$\llbracket f \ x \rrbracket \circ g = \llbracket f \ (x \circ g) \rrbracket$$

Typing issue $\llbracket f \ (x \circ g) \rrbracket :: s \rightarrow (s', a)$, **not** a valid ST applicative expression!

Expand out the ST types and the problem goes away

State fusions

Lemma

(Input state fusion) For all $f : a \rightarrow b$, $x : \text{ST } s' a$ and $g : s \rightarrow s'$:

$$\llbracket f \ x \rrbracket \circ g = \llbracket f \ (x \circ g) \rrbracket$$

Typing issue $\llbracket f \ (x \circ g) \rrbracket :: s \rightarrow (s', a)$, **not** a valid ST applicative expression!

Expand out the ST types and the problem goes away

Lemma

(Output state fusion) For all $f : s \rightarrow s'$, $g : a \rightarrow b$ and $x : \text{ST } s a$:

$$(\text{id} \times f) \circ \llbracket g \ x \rrbracket = \llbracket g \ ((\text{id} \times f) \circ x) \rrbracket$$

What ST does ...

for functions with 2 arguments

What ST does ...

for functions with 2 arguments

Lemma

For all $f : a \rightarrow b \rightarrow c$, $x : \text{ST } s \ a$ and $y : \text{ST } s \ b$:

$$\llbracket f \times y \rrbracket = (\text{uncurry } f \times \text{id}) \circ \text{assoc} \circ (\text{id} \times y) \circ x$$

Inductive step

$$(\text{id} \times \text{from}) \circ \text{label} (\text{Node } l \ r)$$

Inductive step

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Node } l \ r) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Node} (\text{label } l) (\text{label } r) \rrbracket \end{aligned}$$

Inductive step

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Node } l \ r) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Node } (\text{label } l) \ (\text{label } r) \rrbracket \\ = & (\text{id} \times \text{from}) \circ (\text{uncurry Node} \times \text{id}) \circ \text{assoc} \circ (\text{id} \times \text{label } r) \circ \text{label } l \end{aligned}$$

Inductive step

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Node } l \ r) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Node } (\text{label } l) (\text{label } r) \rrbracket \\ = & (\text{id} \times \text{from}) \circ (\text{uncurry Node} \times \text{id}) \circ \text{assoc} \circ (\text{id} \times \text{label } r) \circ \text{label } l \\ = & (\text{uncurry Node} \times \text{id}) \circ (\text{id} \times \text{from}) \circ \text{assoc} \circ (\text{id} \times \text{label } r) \circ \text{label } l \\ = & (\text{uncurry Node} \times \text{id}) \circ \text{assoc} \circ (\text{id} \times ((\text{id} \times \text{from}) \circ \text{label } r)) \circ \text{label } l \\ = & (\text{uncurry Node} \times \text{id}) \circ \text{assoc} \circ (\text{id} \times (\text{label}' \ r \circ \text{from})) \circ \text{label } l \\ = & (\text{uncurry Node} \times \text{id}) \circ \text{assoc} \circ (\text{id} \times \text{label}' \ r) \circ (\text{id} \times \text{from}) \circ \text{label } l \\ = & (\text{uncurry Node} \times \text{id}) \circ \text{assoc} \circ (\text{id} \times \text{label}' \ r) \circ \text{label}' \ l \circ \text{from} \\ = & \llbracket \text{Node } (\text{label}' \ l) (\text{label}' \ r) \rrbracket \circ \text{from} \\ = & \text{label}' (\text{Node } l \ r) \circ \text{from} \end{aligned}$$

State fusions

Lemma

(Input state fusion)

$$\llbracket f \times y \rrbracket \circ g = \llbracket f (x \circ g) y \rrbracket$$

State fusions

Lemma

(Input state fusion)

$$\llbracket f \times y \rrbracket \circ g = \llbracket f (x \circ g) y \rrbracket$$

Lemma

(State shifting)

$$\llbracket f ((\text{id} \times g) \circ x) y \rrbracket = \llbracket f \times (y \circ g) \rrbracket$$

State fusions

Lemma

(Input state fusion)

$$\llbracket f \times y \rrbracket \circ g = \llbracket f (x \circ g) y \rrbracket$$

Lemma

(State shifting)

$$\llbracket f ((\text{id} \times g) \circ x) y \rrbracket = \llbracket f \times (y \circ g) \rrbracket$$

Lemma

(Output state fusion)

$$(\text{id} \times f) \circ \llbracket g \times y \rrbracket = \llbracket g \times ((\text{id} \times f) \circ y) \rrbracket$$

Inductive step revisited

$$(\text{id} \times \text{from}) \circ \text{label} (\text{Node } l \ r)$$

Inductive step revisited

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Node } l \ r) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Node } (\text{label } l) \ (\text{label } r) \rrbracket \end{aligned}$$

Inductive step revisited

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Node } l \ r) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Node} (\text{label } l) (\text{label } r) \rrbracket \\ = & \llbracket \text{Node} (\text{label } l) ((\text{id} \times \text{from}) \circ \text{label } r) \rrbracket \end{aligned}$$

Inductive step revisited

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Node } l \ r) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Node} (\text{label } l) (\text{label } r) \rrbracket \\ = & \llbracket \text{Node} (\text{label } l) ((\text{id} \times \text{from}) \circ \text{label } r) \rrbracket \\ = & \llbracket \text{Node} (\text{label } l) (\text{label}' \ r \circ \text{from}) \rrbracket \end{aligned}$$

Inductive step revisited

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Node } l \ r) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Node} (\text{label } l) (\text{label } r) \rrbracket \\ = & \llbracket \text{Node} (\text{label } l) ((\text{id} \times \text{from}) \circ \text{label } r) \rrbracket \\ = & \llbracket \text{Node} (\text{label } l) (\text{label}' r \circ \text{from}) \rrbracket \\ = & \llbracket \text{Node} ((\text{id} \times \text{from}) \circ \text{label } l) (\text{label}' r) \rrbracket \end{aligned}$$

Inductive step revisited

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Node } l \ r) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Node } (\text{label } l) \ (\text{label } r) \rrbracket \\ = & \llbracket \text{Node } (\text{label } l) \ ((\text{id} \times \text{from}) \circ \text{label } r) \rrbracket \\ = & \llbracket \text{Node } (\text{label } l) \ (\text{label}' \ r \circ \text{from}) \rrbracket \\ = & \llbracket \text{Node } ((\text{id} \times \text{from}) \circ \text{label } l) \ (\text{label}' \ r) \rrbracket \\ = & \llbracket \text{Node } (\text{label}' \ l \circ \text{from}) \ (\text{label}' \ r) \rrbracket \end{aligned}$$

Inductive step revisited

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Node } l \ r) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Node } (\text{label } l) (\text{label } r) \rrbracket \\ = & \llbracket \text{Node } (\text{label } l) ((\text{id} \times \text{from}) \circ \text{label } r) \rrbracket \\ = & \llbracket \text{Node } (\text{label } l) (\text{label}' r \circ \text{from}) \rrbracket \\ = & \llbracket \text{Node } ((\text{id} \times \text{from}) \circ \text{label } l) (\text{label}' r) \rrbracket \\ = & \llbracket \text{Node } (\text{label}' l \circ \text{from}) (\text{label}' r) \rrbracket \\ = & \llbracket \text{Node } (\text{label}' l) (\text{label}' r) \rrbracket \circ \text{from} \end{aligned}$$

Inductive step revisited

$$\begin{aligned} & (\text{id} \times \text{from}) \circ \text{label} (\text{Node } l \ r) \\ = & (\text{id} \times \text{from}) \circ \llbracket \text{Node } (\text{label } l) \ (\text{label } r) \rrbracket \\ = & \llbracket \text{Node } (\text{label } l) \ ((\text{id} \times \text{from}) \circ \text{label } r) \rrbracket \\ = & \llbracket \text{Node } (\text{label } l) \ (\text{label}' \ r \circ \text{from}) \rrbracket \\ = & \llbracket \text{Node } ((\text{id} \times \text{from}) \circ \text{label } l) \ (\text{label}' \ r) \rrbracket \\ = & \llbracket \text{Node } (\text{label}' \ l \circ \text{from}) \ (\text{label}' \ r) \rrbracket \\ = & \llbracket \text{Node } (\text{label}' \ l) \ (\text{label}' \ r) \rrbracket \circ \text{from} \\ = & \text{label}' (\text{Node } l \ r) \circ \text{from} \end{aligned}$$

Conclusions

- Equational proof of correctness on monadic/applicative expression

Conclusions

- Equational proof of correctness on monadic/applicative expression
- Factorisation
- “Loosening” the type system
- Applicative properties of the ST state transformer

Conclusions

- Equational proof of correctness on monadic/applicative expression
- Factorisation
- “Loosening” the type system
- Applicative properties of the ST state transformer
- Similar proof for the prefix-based labelling with Reader monad, without typing problems

Conclusions

- Equational proof of correctness on monadic/applicative expression
- Factorisation
- “Loosening” the type system
- Applicative properties of the ST state transformer
- Similar proof for the prefix-based labelling with Reader monad, without typing problems

... and further work

- Alternative labellings

Conclusions

- Equational proof of correctness on monadic/applicative expression
- Factorisation
- “Loosening” the type system
- Applicative properties of the ST state transformer
- Similar proof for the prefix-based labelling with Reader monad, without typing problems

... and further work

- Alternative labellings
- Traversable

... and further along

- Compiler correctness