State will do

Willem Seynaeve

Simulating one effect with the other

High Level effect

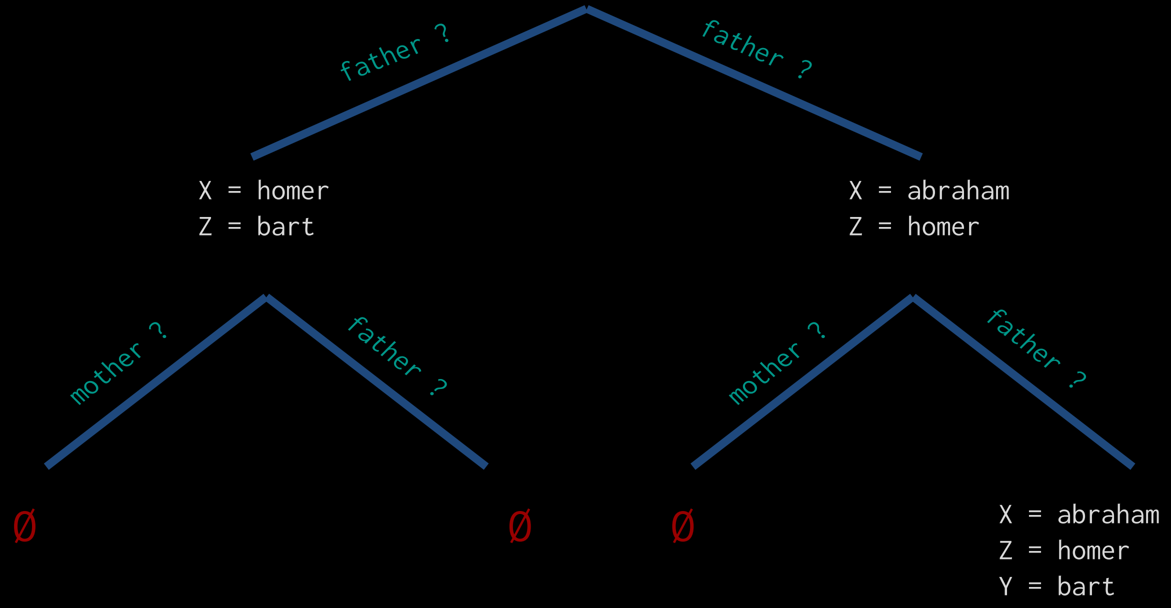
✓ ↓

Low Level effect

Correctness of a Prolog interpreter

```
grandfather(X, Y) :- father(X, Z),
                     (mother(Z, Y) ; father(Z, Y) ).
father(homer, bart).
father(abraham, homer).
```

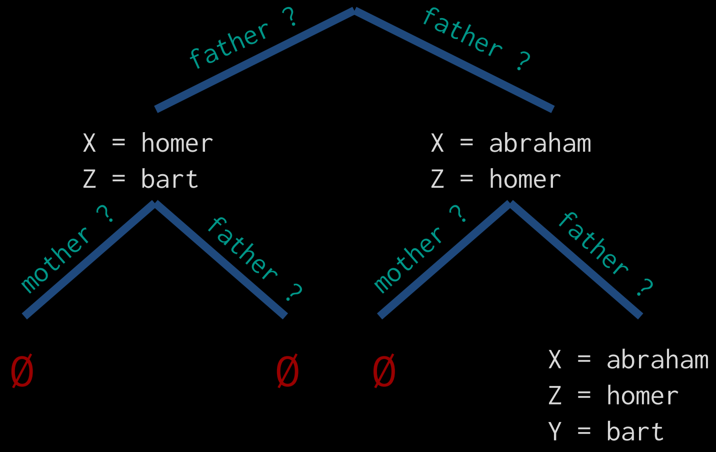
```
?- grandfather(X, Y).
```

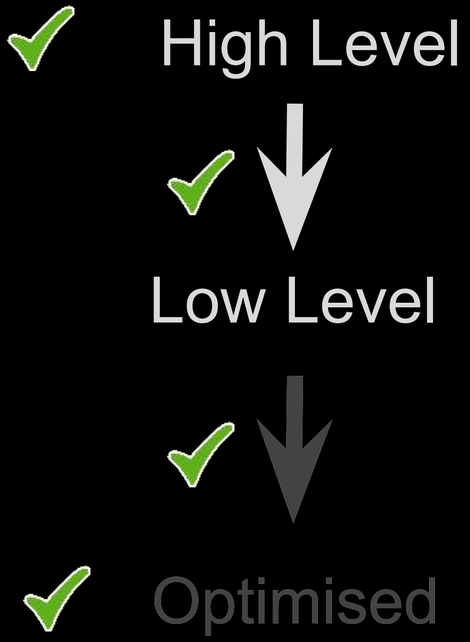


Correctness of a Prolog interpreter

```
grandfather(X, Y) :- father(X, Z),
                     (mother(Z, Y) ; father(Z, Y) ).
father(homer, bart).
father(abraham, homer).
```

```
?- grandfather(X, Y).
```





Equational Reasoning

```
p :: Int  -> Int -> Int
p x y = x+y

m :: Int -> Int
m x = x*x

dist :: Int -> Int -> Int
dist x y = sqrt (x*x + y*y)
```

To Prove:

∀ x y : Int

sqrt $ p (m x) (m y) ≡ dist x y

Proof:

sqrt $ p (m x) (m y)

 ≡ -> *Def of m*

sqrt $ p (x*x) (y*y)

 ≡ -> *Def of plus*

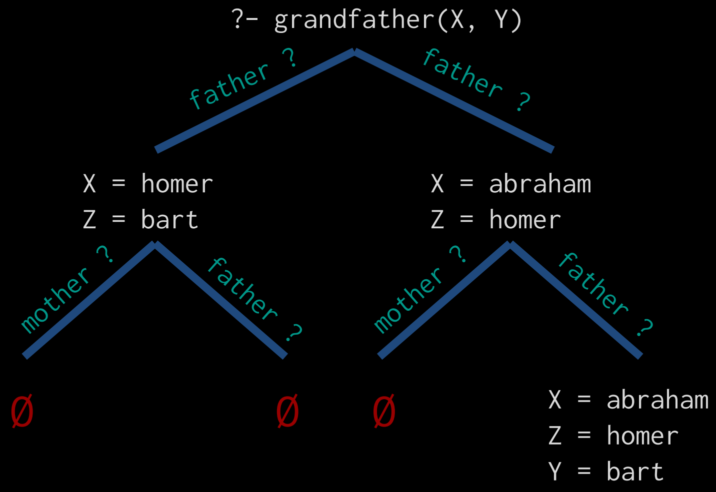
sqrt (x*x + y*y)

 ≡ -> *Def of dist*

dist x y

Correctness of a Prolog interpreter

```
grandfather(X, Y) :- father(X, Z),
                     mother(Z, Y) ; father(Z, Y).
father(homer, bart).
father(abraham, homer).
```



```python
val = 0

def foo(n):
    global val
    val += n
    return val

x = foo(4)

x + x
```

8

$\not\equiv$

```python
val = 0

def foo(n):
    global val
    val += n
    return val

foo(4) + foo(4)
```

12

Monads

The Monad type class:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

The Monad laws

```
--Left identity:
return a >>= f  ≡ f a
--Right identity:
m >>= return    ≡ m
--Associativity:
(m >>= f) >>= g ≡ m >>= (\x -> f x >>= g)
```

Effectful functional programming

```
do
  x <- exp1
  y <- exp2
  return (fun x y)
```

≡

```
exp1 >>= \x ->
  exp2 >>= \y ->
   return (fun x y)
```

Wadler, P.: Monads for functional programming. In: Broy, M. (ed.) Program Design Calculi: Marktoberdorf Summer School. pp. 233–264. Springer-Verlag (1992)

The State Monad

```
class Monad a => State s a | a -> s where
  get :: State s a
  put :: s -> State s a

--The State Laws

put s >>  put s'              = put s'
put s >>  get                 = put s >> return s
get >>= put                   = return ()
get >>= λs -> get >>= k s = get >>= λs -> k s s
```

To prove:

```
get >>= \s -> put s >>= \_ -> get >>= \a -> return (a+2)
≡
get >>= \a -> return (a+2)
```

Proof:

```
get >>= \s -> put s >>= \_ -> get >>= \a -> return (a+2)
≡ get-put law
return () >>= \_ -> get >>= \a -> return (a+2)
≡ monad law
( \_ -> get >>= \a -> return (a+2) ) ()
≡ beta reduction
get >>= \a -> return (a+2)
```

```
do
  s <- get
  put s
  a <- get
  return (a+2)
```

≡

```
do
  a <- get
  return (a+2)
```

Stateful Program

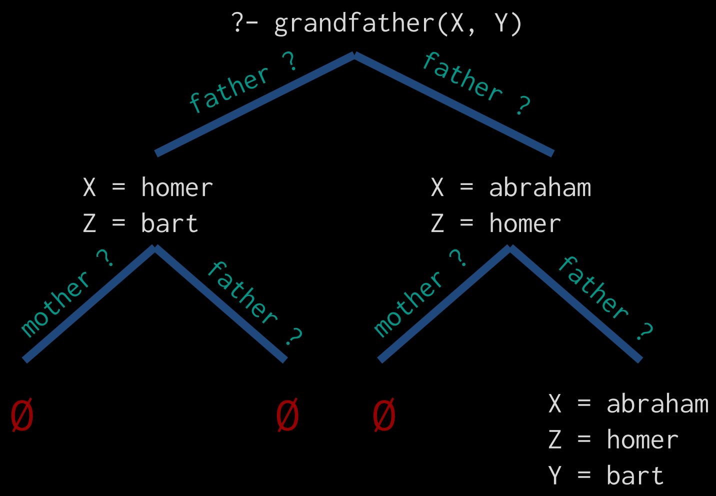
```python
val = 0

def foo(n):
    global val
    val += n
    return val
```

```haskell
foo :: Int -> State Int Int
foo n = do
  val <- get
  put (val + n)
  ret <- get
  return ret
```

Correctness of a Prolog interpreter

```
grandfather(X, Y) :- father(X, Z),
                     mother(Z, Y) ; father(Z, Y).
father(homer, bart).
father(abraham, homer).
```



Nondeterminism

The Nondet class:

```
class Monad m => MonadNondet m where
   fail :: m a
   comb :: m a -> m a -> m a
```

Example : Sudoku

Gibbons, J., Hinze, R.: Just do it: simple monadic equational reasoning. In: Danvy, O. (ed.) International Conference on Functional Programming. pp. 2–14. ACM Press (2011)

Nondeterminism

	8	4	1		5		6	
3		9		4		2		1
		5	9					8
		7	2			6		
6		1				9		2
		2			3	1		
2					8	5		
5		6		7		8		4
	1		6		9	7	2	

$\Rightarrow$

7	8	4	1		5		6	
3		9		4		2		1
		5	9					8
		7	2			6		
6		1				9		2
		2			3	1		
2					8	5		
5		6		7		8		4
	1		6		9	7	2	

Nondeterminism

7	8	4	1		5		6	
3		9		4		2		1
		5	9					8

7	8	4	1	2	5		6	
3		9		4		2		1
		5	9					8

7	8	4	1	3	5		6	
3		9		4		2		1
		5	9					8

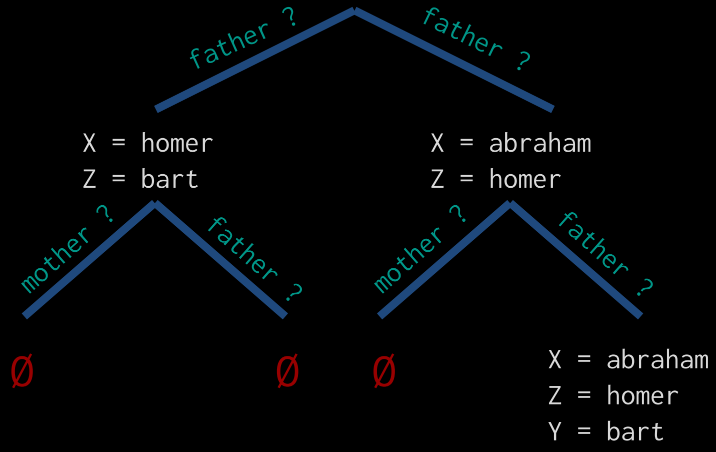
Combining effects

```
class Monad a => StateNondet s a | a -> s where
   fail   :: StateNondet s a
   (comb) :: StateNondet s a -> StateNondet s a -> StateNondet s a
   get    :: StateNondet s a
   put    :: s -> StateNondet s a
```

Correctness of a Prolog interpreter

```
grandfather(X, Y) :- father(X, Z),
                     mother(Z, Y) ; father(Z, Y).
father(homer   , bart).
father(abraham, homer).

        ?- grandfather(X, Y)
```



High-Level

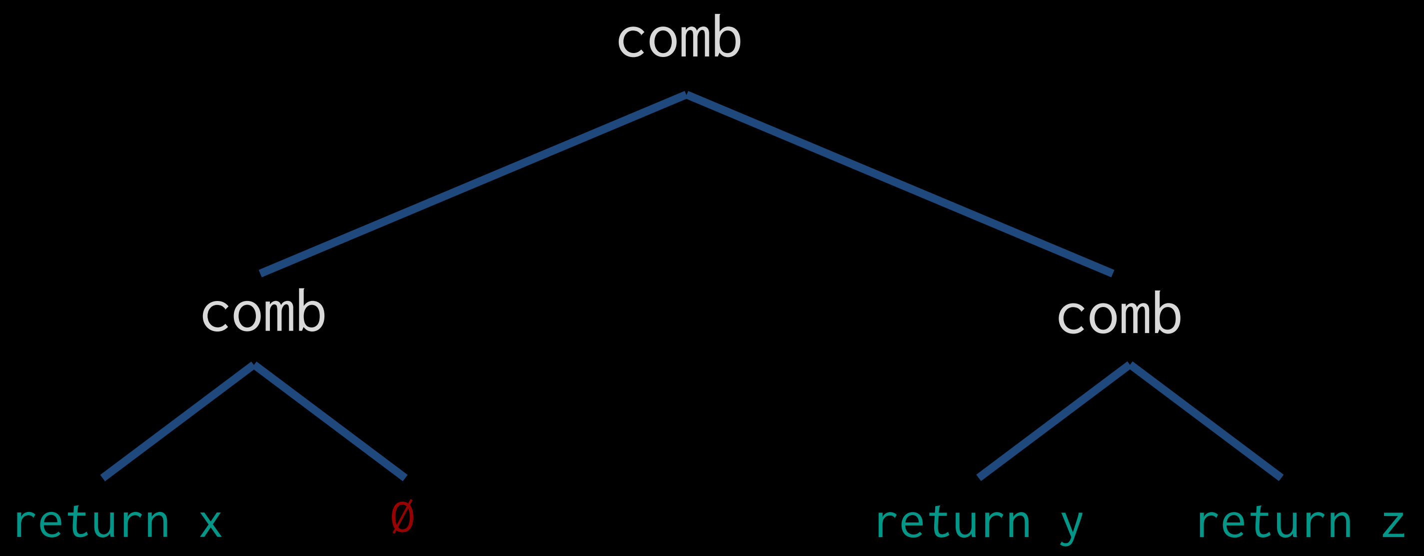
Low-Level

Optimised

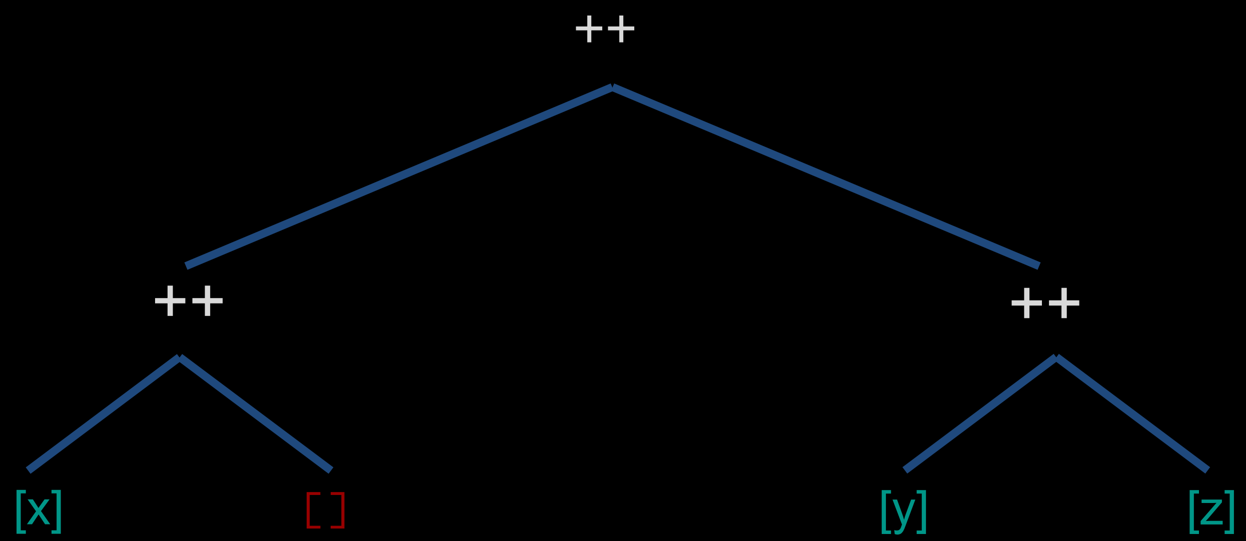
Example simulation

$$\begin{tikzcd} \text{Nondet} \arrow[d] \\ \text{State} \end{tikzcd}$$

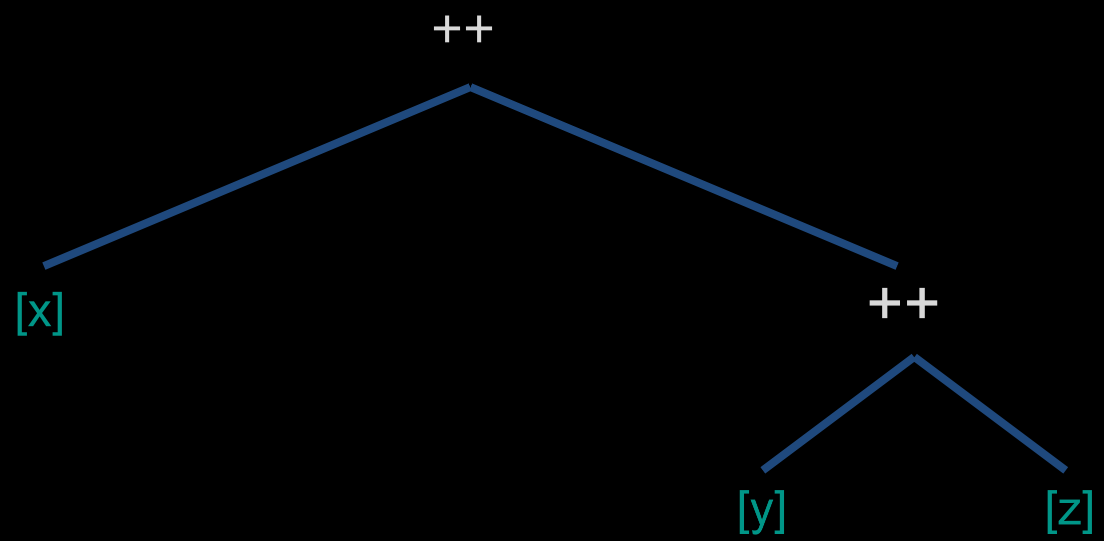
Example simulation



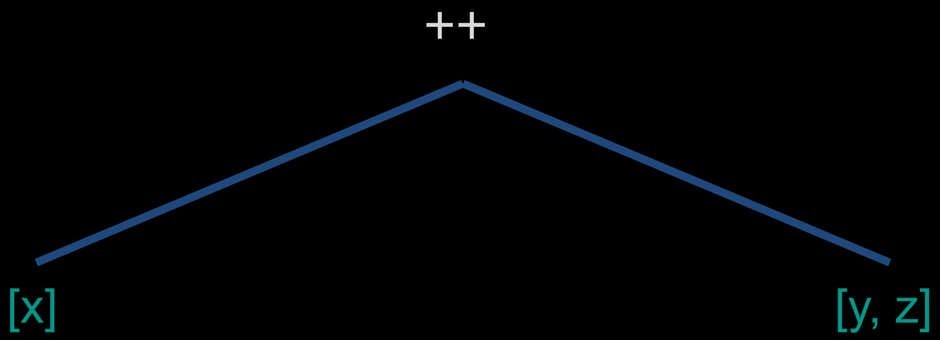
The [ ] implementation



The [] implementation



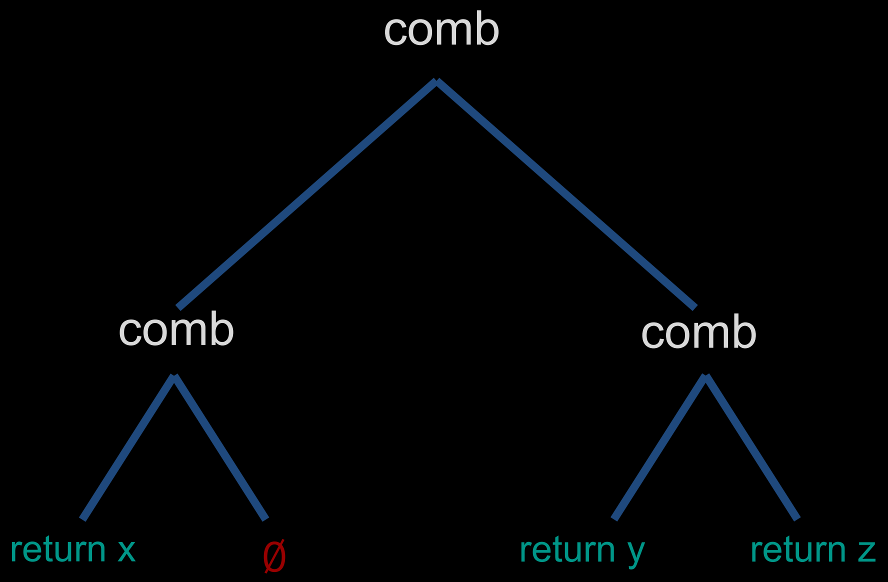
The [ ] implementation



The [ ] implementation

[x, y, z]

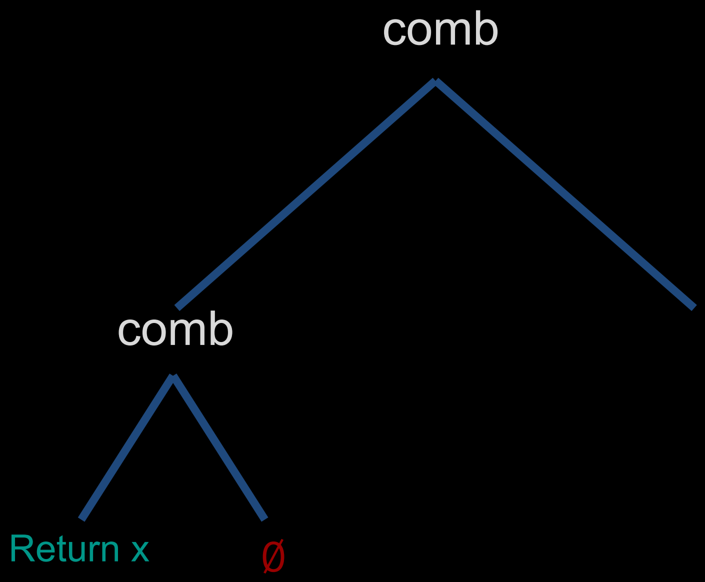
Nondet with State



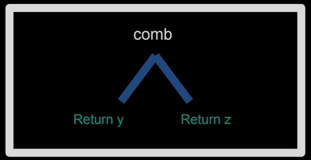
Stack

Solutions

Nondet with State

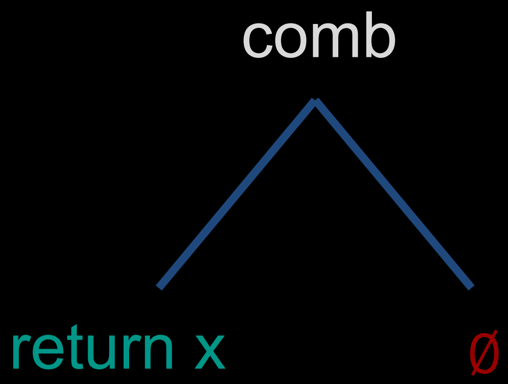


Stack

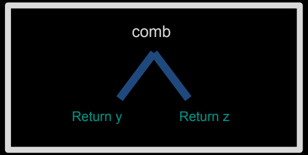


Solutions

Nondet with State

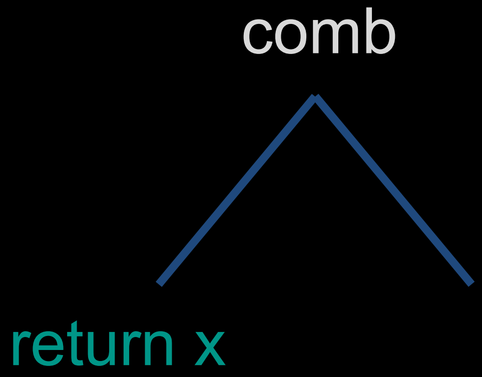


Stack

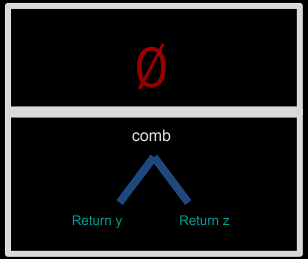


Solutions

Nondet with State



Stack

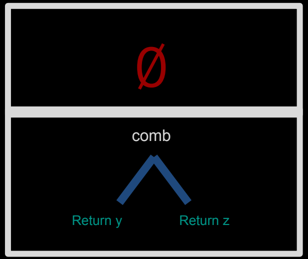


Solutions

Nondet with State

return x

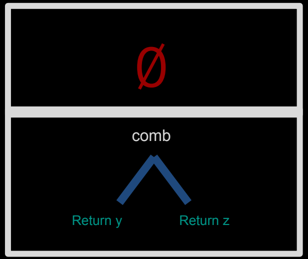
Stack



Solutions

Nondet with State

Stack



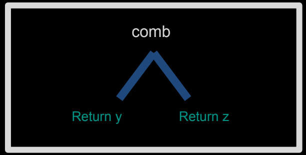
Solutions

[x]

Nondet with State

Stack

∅

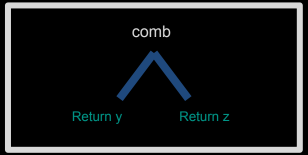


Solutions

[x]

Nondet with State

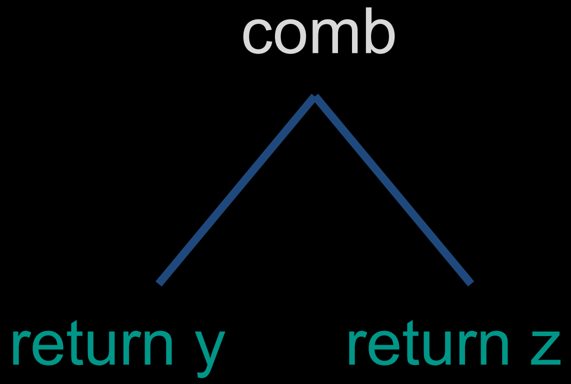
Stack



Solutions

[x]

Nondet with State

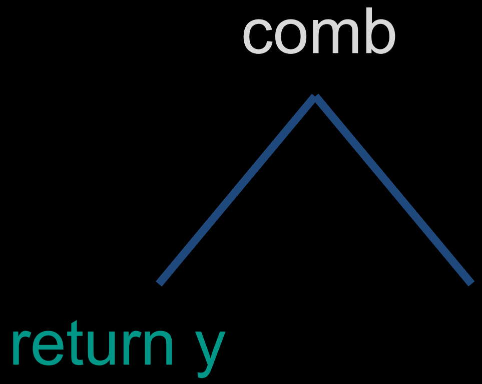


Stack

Solutions

[x]

Nondet with State



Stack

$\boxed{\text{return z}}$

Solutions

[x]

Nondet with State

return y

Stack

$\boxed{\text{return z}}$

Solutions

[x]

Nondet with State

Stack

$\boxed{\text{return z}}$

Solutions

[x, y]

Nondet with State

Stack

return z

Solutions

[x, y]

Nondet with State

Stack

Solutions

[x, y, z]

Conclusions

- Simulate one effect with the other

Nondet

✓ ↓

State

State + Nondet

✓ ↓

State

- Equational reasoning about effectful programs
- Mechanised proof (coq)

Questions?

References

- Wadler, P.: Monads for functional programming. In: Broy, M. (ed.) Program Design Calculi: Marktoberdorf Summer School. pp. 233–264. Springer-Verlag (1992)
- Gibbons, J., Hinze, R.: Just do it: simple monadic equational reasoning. In: Danvy, O. (ed.) International Conference on Functional Programming. pp. 2–14. ACM Press (2011)