Advances in Programming Languages

APL10: State Transformers

Ian Stark

School of Informatics
The University of Edinburgh

Thursday 11 February
Semester 2 Week 5

Foreword

Some Types in Haskell

This is the last of four lectures about some features of types and typing in Haskell types, specifically:

- Type classes
- Multiparameter type classes, constructor classes,
- Monads and interaction with the outside world
- Encapsulating stateful computation

Foreword

Some Types in Haskell

This is the last of four lectures about some features of types and typing in Haskell types, specifically:

- Type classes
- Multiparameter type classes, constructor classes,
- Monads and interaction with the outside world
- Encapsulating stateful computation

Preview

In this lecture we shall see two applications of monads, one a refinement of the other.

- The State monad from the Control.Monad.State library.

  This makes it possible to describe sequential stateful computation within a lazy functional language.

- The ST and its runST operation from Control.Monad.ST

  This allows more general stateful computation, allocating and updating storage, with efficient in-place implementation; but encapsulated so that it becomes indistinguishable from purely functional code.

Outline

There's a place for state

Purely functional languages are able to express many algorithms in concise and modular fashion.

However, there remain some cases where known efficient (and convenient) algorithms rely on *mutable state* of some kind:

- Memoization, incrementally-modified hash tables;
- Union/find;
- Graph traversal and manipulation.

For these, we would like some way to:

- Naturally express the imperative algorithm in Haskell;
- Wrap it up so it can be combined with other, purely functional, components.

The State monad

```
module Control.Monad.State

newtype State s a = State { runState :: (s -> (a, s)) }
```

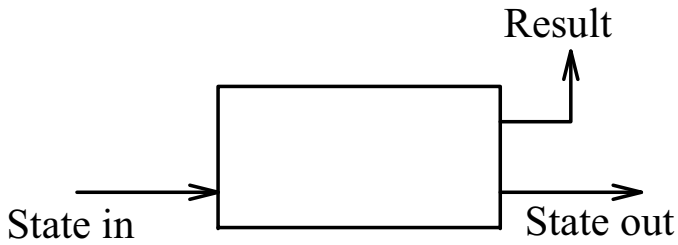
A value of type "State s a" represents a computation which manipulates a state component of type "s" to return a result of type "a".

The State monad

module Control.Monad.State

newtype State s a = State { runState :: (s -> (a, s)) }

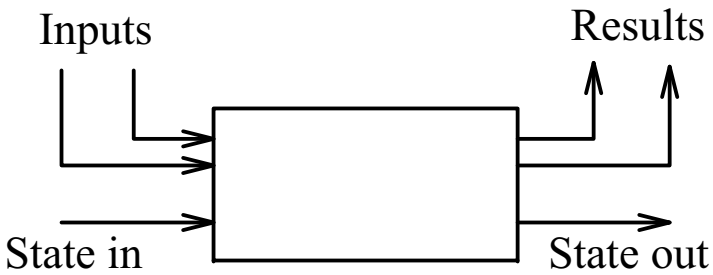
A value of type "State s a" represents a computation which manipulates a state component of type "s" to return a result of type "a".



The State monad

```
module Control.Monad.State

newtype State s a = State { runState :: (s -> (a, s)) }
```

A value of type "State s a" represents a computation which manipulates a state component of type "s" to return a result of type "a".

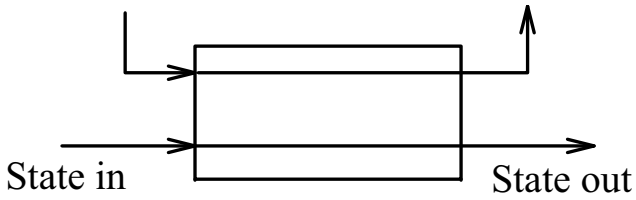


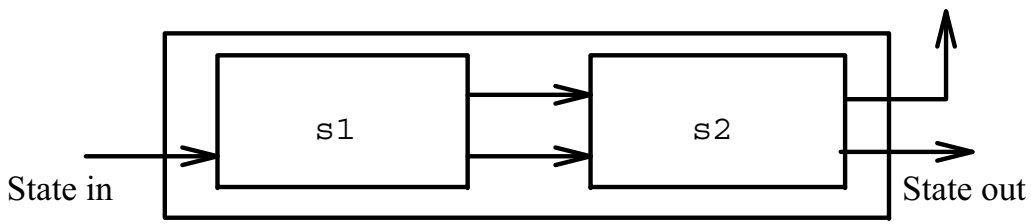
A stateful computation taking inputs and returning results would have a type like "Bool -> String -> State s (Int,Int)".

Monad operations

```
module Control.Monad.State

newtype State s a = State { runState :: (s -> (a, s)) }

instance Monad (State s a) where
return :: a -> State s a
(>>=) :: State s a -> (a -> State s b) -> State s b
```

The monad "return" simply gives back the value, leaving the state unchanged; and "(>>=)" sequences two stateful computations, passing the output state and result of the first as input to the second.





Effects

```
module Control.Monad.State

newtype State s a = State { runState :: (s -> (a, s)) }

get :: State s s                        -- Fetch contents of state
put :: s -> State s ()                  -- Update contents

modify :: (s -> s) -> State s ()        -- Modify contents

evalState :: State s a -> s -> a        -- Run to result
execState :: State s a -> s -> s        -- Run to final state
```

Example

```
import Control.Monad.State

data Tree a = Leaf a | Node (Tree a) (Tree a) deriving (Show, Eq)

number :: Tree a -> Tree Int          -- Number the leaves of
number t = evalState (walk t) 0       -- a tree from left to right

walk :: Tree a -> State Int (Tree Int)  -- Number using state

walk (Leaf _)   = do i <- get
                     put (i+1)
                     return (Leaf i)

walk (Node l r) = do l' <- walk l
                     r' <- walk r
                     return (Node l' r')
```

Outline

Source paper

J. Launchbury and S. L. Peyton Jones.
Lazy functional state threads.
In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, SIGPLAN Notices, 29(6), pages 24–35. ACM Press, June 1994.

The "State" monad makes it possible to write stateful computations in an imperative style, but does not enforce this: code might duplicate or reset the state.

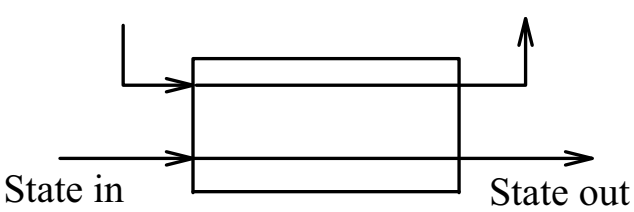
This paper showed how to build a richer "ST" (state transformer) monad which supports more operations, while guaranteeing that the state is single-threaded, allowing for general imperative algorithms *and* efficient implementation.

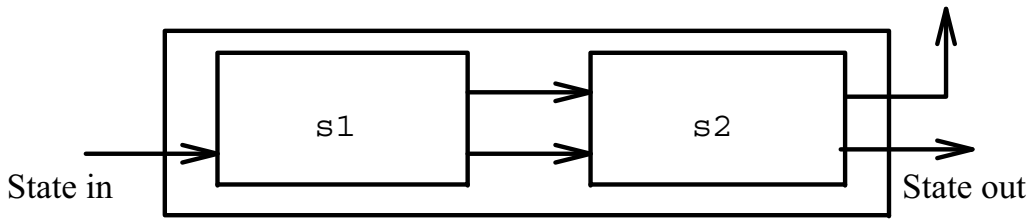
The ST monad is a State monad

module Control.Monad.ST

data ST s a

instance Monad (ST s a) **where**
return :: a -> State s a
(>>=) :: State s a -> (a -> State s b) -> State s b





Mutable variables in the ST monad

```
module Control.Monad.ST

data ST s a

newVar  :: a -> ST s (MutVar s a)
readVar :: MutVar s a -> ST s a
writeVar :: MutVar s a -> a -> ST s ()
```

A value of type "MutVar s a" is a reference to a mutable variable, storing type "a" in a state of type "s".

Operation "newVar" creates and initialises a fresh variable, "readVar" reports its current value, and "writeVar" updates it.

Any state thread can create any number of "MutVar" variables.

Example with mutable variables

```
import Control.Monad.ST

swapVar :: MutVar s a -> MutVar s a -> ST s ()

swapVar p q = do x <- readVar p
                 y <- readVar q
                 writeVar p y
                 writeVar q x
                 return ()
```

Here "swapVar" will take two variables return the computation that exchanges their contents.

Notice that the variables must share the same state type "s".

Running a state transformer

```
module Control.Monad.ST

runST :: ST s a -> a    -- Can we do this?
```

We might try to write an operation that creates a fresh state of type "s", and then runs a computation within that. But there is a problem.

Running a state transformer

```
module Control.Monad.ST

runST :: ST s a -> a    -- Can we do this?
```

We might try to write an operation that creates a fresh state of type "s", and then runs a computation within that. But there is a problem.

```
let v = runST (newVar True) -- Create a fresh variable here ...
in
runST (readVar v)           -- ... best not to try and read it here
```

A better type for runST

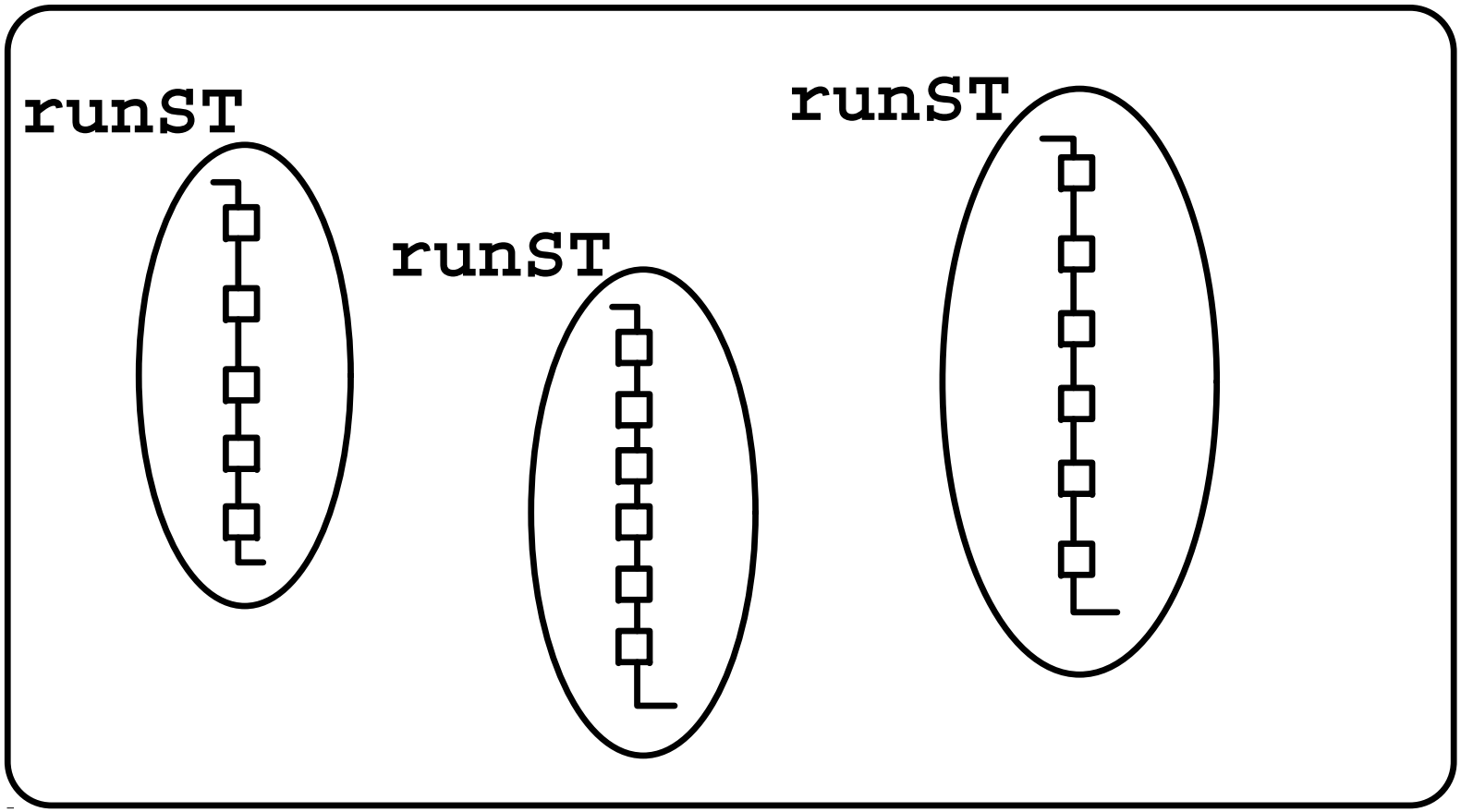
```
module Control.Monad.ST

runST :: (forall s. ST s a) -> a   -- Rank 2 polymorphism
```

The solution is to require that "runST" is only ever applied to computations that are polymorphic in the state type — they don't require any specific structure in the state, or export it.

An ST-computation can create fresh variables, read and write them, but everything must stay local.

Higher rank polymorphism is an extension to regular Haskell: types like this can be checked, but in general not inferred.

Isolated state threads



Arrays in ST

```
data MutArray s i e

newArr   :: Ix i => (i,i) -> e -> ST s (MutArray s i e)

readArr  :: Ix i => MutArray s i e -> i -> ST s e

writeArr :: Ix i => MutArray s i e -> i -> e -> ST s ()

freezeArr :: Ix i => MutArray s i e -> ST s (Array i e)
```

Mutable variables and imperative arrays are enough to implement classic imperative algorithms, both simple and complex.

Externally, code that uses "ST s a" and "runST" can be used exactly as it was purely functional, and the polymorphic typing requirements of "runST" ensure that its state will neither escape nor be disrupted by other code.

Arrays in ST

```
data MutArray s i e

newArr   :: Ix i => (i,i) -> e -> ST s (MutArray s i e)

readArr  :: Ix i => MutArray s i e -> i -> ST s e

writeArr :: Ix i => MutArray s i e -> i -> e -> ST s ()

freezeArr :: Ix i => MutArray s i e -> ST s (Array i e)
```

Because type "ST s a" is only accessible through these effects, it is certain that the state component "s" will be single-threaded: which allows for efficient implementation and in-place update.

What is more, this can be handled entirely by standard compiler rewriting optimisations: passing a single token around will ensure data dependencies are respected; yet the actual variables and arrays can be allocated in memory and updated in place. We even get lazy evaluation of these state threads.

IO is ST

```
data RealWorld                    -- No constructors, entirely abstract

type IO a = ST RealWorld a -- IO computations act on the real world
```

This is exactly how the "IO" monad is implemented: with an entirely abstract state type, whose concrete realisation is the external world.

Because "IO a" specifies a specific state type "RealWorld", there is no possibility of applying "runST", whose argument must be state-independent.

This is good, because Haskell has no operation to create or destroy the "RealWorld" at will. Instead, the whole program has type "IO a", and it is run in the actual "RealWorld".

Outline

Summary

In this lecture we saw two applications of monads, one a refinement of the other.

- The State monad from the Control.Monad.State library.

  This makes it possible to describe sequential stateful computation within a lazy functional language.

- The ST and its runST operation from Control.Monad.ST

  This allows more general stateful computation, allocating and updating storage, with efficient in-place implementation; but encapsulated so that it becomes indistinguishable from purely functional code.

Homework

Read the following paper, describing runST, its applications and implementation.

J. Launchbury and S. L. Peyton Jones.
Lazy functional state threads.
In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, SIGPLAN Notices, 29(6), pages 24–35. ACM Press, June 1994.

Also available in a longer version, with additional examples:

J. Launchbury and S. L. Peyton Jones.
State in Haskell.
LISP and Symbolic Computation, 8(4):293–342, December 1995.