Concurrency Monads

Tuesday, October 25, 2011
13:29

Functional Pearls

→ Beautiful Ideas explained Beautifully

→ Category Theory

Monads:

haskell allows class overloading using type inference.

```
class Monad m
    where
        (*) :: m α → (α → m b) → m b
        return :: α → m α
```

"bind" do an operation with? on? α to get b

shove something into the monad

Monad: a way to represent a computation that produce a result

Haskell is purely functional. There are no side effects.

∃ monad called IO in Haskell to do syscalls, etc

The writer monad

```
class Monad m ⇒ Writer m where
    write :: String → m ()
```

Monad
↑
Writer

"A monad m is a writer if its has a write function?"

J. Functional Programming 9 (3): 313–323, May 1999. Printed in the United Kingdom

screenshots from this paper

FUNCTIONAL PEARL

A poor man's concurrency monad

KOEN CLAESSEN
Chalmers University of Technology
(*e-mail:* koen@cs.chalmers.se)

```
type W α = (α, String)

instance Monad W where
  (a, s) ⋆ k = let (b, s') = k a in (b, s ++ s')
  return x = (x, "")

instance Writer W where
  write s = ((), s)
```

strings are lists of chars

list concat operator

takes any Wα, gives us a String

```
output        :: W α → String
output (a, s) = s
```

You can do cool Monad transforms in Haskell.

```
class MonadTrans τ where
  lift :: Monad m ⇒ m α → (τ m) α
```

$$\texttt{type C } m\ \alpha = (\underbrace{\alpha \to \texttt{Action } m}_{\text{continuation (that can consume } \alpha)}) \to \underbrace{\texttt{Action } m}_{\text{final result}}$$

```
data Action m
  = Atom (m (Action m))
  | Fork (Action m) (Action m)
  | Stop
```

either

→ do an atomic step

→ fork yourself

→ stop

```
instance Monad m ⇒ Monad (C m) where
  f ⋆ k      = λc. f (λa. k a c)   = λc : (b → Am)
  return x   = λc. c x
```

$f \in (\alpha \to \overset{\text{Action}}{Am}) \to Am$

$k \in \alpha \to ((B \to Am) \to Am)$

$f(\lambda a : a\ k\ a\ c)$

Haskell doesn't explicitly check something w/ bind & return

⇒ something about the halting problem

⇒ implicit

```
atom :: Monad m ⇒ m α → C m α
atom p = λc. Atom (do a ← p
                      ; return (c a))
```

has type m of α

Do notation

```
expr1 ⋆ λx.                    do x ← expr1
expr2 ⋆ λ_.                      ; expr2
expr3 ⋆ λy.                      ; y ← expr3
return expr4                     ; return expr4
```

In addition, we have a function that uses the `Stop` constructor, called `stop`. It discards any continuation, thus ending a computation.

```
stop :: Monad m ⇒ C m α
stop = λc. Stop
```

To access `Fork`, we define two operations. The first, called `par`, combines two computations into one by forking them both, and passing the continuation to both parts. The second, `fork`, resembles the more traditional imperative fork. It forks its argument after turning it into an action, and continues by passing () to the continuation.

```
par        :: Monad m ⇒ C m α → C m α → C m α
par m1 m2  = λc. Fork (m1 c) (m2 c)

fork       :: Monad m ⇒ C m α → C m ()
fork m     = λc. Fork (action m) (c ())
```

action m = m (λ a. stop)

do this. i.e run then stop

continue on c, which is what the env gave us.

```
type C m α = (α → Action m) → Action m
round :: Monad m ⇒ [Action m] → m ()
round []       = return ()
round (a : as) = case a of
  Atom am      → do a' ← am ; round (as ++ [a'])
```

do something then a' := residual actions

Action m sub-type

append and recurse

```
round :: Monad m ⇒ [Action m] → m ()
round []       = return ()
round (a : as) = case a of
  Atom a_m     → do a' ← a_m ; round (as ++ [a'])
  Fork a_1 a_2 → round (as ++ [a_1, a_2])
  Stop         → round as
```

do something then a' := residual actions

Action m sub-type

append and recurse

Monad ⇒ you're constructing a program which you run at the end

⇒ the monad transforms the program into what you want

```
instance Writer m ⇒ Writer (C m) where
  write s = lift (write s)
```

defining for writer(C m)

from writer m

from the fact that writer (C m) is a monad transformer

```
loop   :: Writer m ⇒ String → m ()
loop s = do write s ; loop s
```

```
example  :: Writer m ⇒ C m ()
example  = do write 'start!'
              ; fork (loop 'fish')
              ; loop 'cat'
```

spawns loop 'cat' in parallel

The result of the expression `output (run example)` looks like the following string:

```
'start!fishcatfishcatfishcatfishcatfishcatfishca ...'
```

Because we defined `write` as an atomic action, the writing of one '`fish`' and one '`cat`' cannot interfere. If we want finer grained behaviour, we can split one write action into several write actions, e.g. the separate characters of a string. A simple way of doing this is to change the lifting of `write`:

```
instance Writer m ⇒ Writer (C m) where
  write []     = return ()
  write (c : s) = do lift (write [c]) ; write s
```

The lifting is now done character-by-character. The result of the expression `output (run example)` now looks like this.

```
'start!fciasthcfaitschafticsahtfciasthcfaitscha ...'
```