Concurrent Monads
Tuesday, October 25, 2011
13:29

**Functional Pearls**

Concurrent Monads: Haskell allows classEverybody uses type inference.

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

**Category Theory**

Concurrent Monads: a way to represent a computation that produces a result.

Haskell is purely functional. There are no side effects.

I invented the IO monad to do exactly that.

**The Writer Monad**

```
class Monad m => Writer m where
    runWriter :: Writer a b -> m (a, b)
n```

"A monad is a writer if it has a write function?"

---


**FUNCTIONAL PEARL**

*A poor man's concurrency monad*

KOEN CLAESSEN
Chalmers University of Technology (e-mail: koen@ca.chalmers.se)
type \( W x = (x, \text{String}) \)

instance Monad \( W \) where
\[
(a, s) * k = \text{let } (h, s') = k a \text{ in } (h, s ++ s')
\]
return \( x = (x, '') \)

instance Writer \( W \) where
\[
\text{write } s = ((), s)
\]

output :: \( W x \rightarrow \text{String} \)
output \((a, s) = s \)

You can do cool Monad transformers in Haskell.

class MonadTrans \( \tau \) where
\[
\text{lift :: Monad } m \rightarrow m x \rightarrow (\tau m) x
\]

type \( C m x = (x \rightarrow \text{Action } m) \rightarrow \text{Action } m \)

\[
\text{data Action } m \text{ where}
\]
\[
\begin{align*}
\text{Atom } & (m (\text{Action } m)) \\
\text{Fork } & (\text{Action } m) (\text{Action } m) \\
\text{Stop} & \\
\end{align*}
\]

data Action \( m \)
\[
\begin{align*}
\text{data Action } m \text{ where}
\end{align*}
\]

data Action \( m \)
\[
\begin{align*}
\text{data Action } m \text{ where}
\end{align*}
\]

instance Monad \( m \rightarrow \text{Monad (C m)} \) where
\[
\lambda c. f (\lambda a. k a c) = c = (\lambda b. a m)
\]
return \( x = \lambda c. c x \)

\[
\begin{align*}
\text{instance Monad } m \rightarrow \text{Monad (C m)} \text{ where}
\end{align*}
\]

\[
\begin{align*}
\text{instance Monad } m \rightarrow \text{Monad (C m)} \text{ where}
\end{align*}
\]

\[
\begin{align*}
\text{instance Monad } m \rightarrow \text{Monad (C m)} \text{ where}
\end{align*}
\]

\[
\begin{align*}
\text{instance Monad } m \rightarrow \text{Monad (C m)} \text{ where}
\end{align*}
\]

\[
\begin{align*}
\text{instance Monad } m \rightarrow \text{Monad (C m)} \text{ where}
\end{align*}
\]
Haskell doesn't explicitly check安全性 with `bind` & `return` → soundly about the unity problem

atom :: Monad m → m α → C m α

atom ρ = λ c. Atom (do a ← ρ
    ; return (c a))

Do notation

expr_1 * i.x.  do x ← expr_1
expr_2 * i._   ; expr_2
expr_3 * i.y.  ; y ← expr_3
return expr_4  ; return expr_4

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 x
stop = λ c. 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 (λ) to the continuation.

par :: Monad m ⇒ C m x → C m x
par m_1 m_2 = λ c. Fork (m_1 c) (m_2 c)

fork :: Monad m ⇒ C m x → C m ()
fork m = λ c. Fork (action m) (c ())

action m = m (λ a, stop)

round :: Monad m ⇒ [Action m] → m ()
round [] = return ()
round (a : as) = case a of
    Atom a_m → do a' ← a_m ; round (as ++ [a'])
    Action m a_s → do something with a_s then round a_s

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 a1 a2 → round (as ++ [a1, a2])
  Stop → round as

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

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'

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 ...'