A Poor Man's Concurrency Monad

Koen Claessen

without adding primitives, we construct a way to lift any monad into a limited, but useful concurrent setting.

Monads

- abstraction from computation

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

- we use special notation

```
do a <- expr1          expr1 >>= \a ->
   expr2               expr2 >>= \_ ->
   b <- expr3          expr3 >>= \b ->
   expr4               expr4
```

Writer Monad

- can produce some output during computation

```
class Monad m => Writer m
  where
    write :: String -> m ()
```

- An implementation could be:

- type W a = (a, String)

- instance Monad W where

```
m >>= k = let (a, s) = m
              (b, s') = k a
          in (b, s++s')
return a = (a, "")
```

- instance Writer W where

```
write s = ( (), s)
```

- output :: W a -> String

```
output (a, s) = s
```

Monad Transformer

- adds a feature to an existing monad

```
class MonadT t where
  lift :: Monad m
       => m a -> (t m) a
```

- examples:
  - state
  - exception
  - non determinism
- "compose your own monad" - LEGO

Concurrency

* interleaving actions
* atomic actions are actions in some monad
* round robin scheduler

---

* process has to consist of initial action + future.

Actions

We build actions from three different constructions:
atomic actions, forked actions and no-action.

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

We use constructors:

- general & simple
- expressive

See also Scholz [2].

Continuation

specifies what to do with result.

```
type C a =
  (a -> Action) -> Action
```

parametrize over a monad:

```
type C m a =
  (a -> Action m) -> Action m
```

for some type Action that stands for a process.

It is a monad:

```
instance Monad (C m) where
  m >>= k = \cont -> m
                     (\a -> k a cont)
  return a = \cont -> cont a
```

Useful Operations

Some functions that make life easier.

- Turn a `C m a` into an `Action`:

```
action :: C m a -> Action m
action c = c (\a -> Stop)
```

- Turn an `m a` into an (atomic) `C m a`:

```
atom :: m a -> C m a
atom m = \cont ->
  Atom (do a <- m
           return (cont a))
```

- End a process (the empty process):

```
stop :: C m a
stop = \cont -> Stop
```

Fork

Some operations on fork:

- 'Imperative' fork:

```
fork :: C m a -> C m ()
fork c = \cont -> Fork
  (action c) (cont ())
```

- 'Alegebraic' or symmetrical fork:

```
par :: C m a -> C m a -> C m a
par c1 c2 = \cont ->
  Fork (c1 cont) (c2 cont)
```

Running a C

Ideally, we would like a function

```
run :: C m a -> m a
```

this is "not" possible, due to typing problems.
We will define a function

```
run :: C m a -> m ()
```

This means we'll only get the side-effects of the computation.

Round Robin

simple scheduler.

```
round :: [Action m] -> m ()
round []     = return ()
round (p:ps) =
   case p of
   -Atom ma ->
        do p' <- ma
           round (ps ++ [p'])
   -Fork p1 p2 ->
        round (ps ++ [p1, p2])
   -Stop ->
        round ps
```

Using C

- We can use the scheduler to define:

```
run :: C m a -> m ()
run c = round [action c]
```

- We can construct C's with atom, fork, stop,

  and can run them using run .

C is a Monad Transformer

C can be made an instance of MonadTrans.

```
instance MonadTrans C
  where
    lift = atom
```

All lifted actions become atomic actions in the new setting.

Example 1: Writer

We lift every writer monad:

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

Every write action is now atomic.

```
example :: C W ()
example = do write "hej!"
             fork (loop "apa")
             fork (loop "hund")
  where
    loop s = do write s
                loop s
```

will result in:

[redacted]

hej! apa hund apa hund apa ....

Example 2: Another lifting

We can lift writers in a different way:

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

a write action is now split up in atomic actions for each character.

hej! ahpuanadphaupn....