Partiality is an Effect

Venanzio Capretta

Thorsten Altenkirch

Tarmo Uustalu

WG 2.8, Kalvi, 1–4 Oct. 2005

Outline

- The problem of partiality
- The delay datatype and monad
- Constructive domain theory and recursion
- Combining partiality with other effects

Problem

- Partial/non-terminating programs may be ok, but partial proofs are not.
- In dependently typed programming, where no distinction is made between programs and proofs, this becomes critical.
- Partial maps (total functions on subsets) are no solution.
- Idea: Could non-termination be seen as an effect?
- Answer: Yes! Just use the fact that termination/non-termination is about waiting.

Delay datatype

- Delayed values:

```
data Delay a = Now a | Later (Delay a)        -- coinductive
```

- Infinite delay:

```
never :: Delay a
never = Later never
```

- Minimalization:

```
minim :: (Int -> Bool) -> Delay Int
minim p = minimFrom p 0

minim :: (Int -> Bool) -> Int -> Delay Int
minimFrom p n = if p n then Now n else Later (minimFrom p (n+1))
```

- Competition of two computations:

```
race :: Delay a -> Delay a -> Delay a
race (Now a) _ = Now a
race (Later _) (Now a) = Now a
race (Later d) (Later d') = Later (race d d')
```

- Competition of omega many computations:

```
omegarace :: [Delay a] -> Delay a
omegarace (d:ds) = race d (Later (omegarace ds))
```

- Note that all function definitions above are guarded corecursive.

MONADIC STRUCTURE OF DELAY

- Delay is a monad: we have the Kleisli identity and composition:

```
instance Monad Delay where
    return = Now
    Now a   >>= k = k a
    Later d >>= k = Later (d >>= k)
```

- A special operation:

```
repeat :: (a -> Delay (Either b a)) -> a -> Delay b
repeat k a = do v <- k a
                case v of
                    Left  b -> return b
                    Right a -> Later (rep k a)
```

This is not guarded in an obvious fashion.

- A closely related one with a clearly guarded definition:

```
while :: (a -> Delay (Either b a)) ->
                       Delay (Either b a) -> Delay b
while k (Now (Left b))  = Now b
while k (Now (Right a)) = Later (while k (k a))
while k (Later c)       = Later (while k c)

repeat :: (a -> Delay (Either b a)) -> a -> Delay b
repeat k a = while k (Now (Right a))
```

- More specific versions:

```
repeat' :: (a -> Delay a) -> (a -> Delay Bool) -> a -> Delay a
repeat' k p
   = repeat (\ a -> do a' <- k a
                       b  <- p a'
                       return (if b then Left a' else Right a'))

while' :: (a -> Delay Bool) -> (a -> Delay a) -> a -> Delay a
while' p k
   = repeat (\ a -> do b <- p a
                       if b then do { a' <- k a ; return (Right a') }
                            else return (Left a))
```

Some category theory

- Monads like the delay monad $A \mapsto \nu X.A + X$ have been discussed extensively by Adamek et al in category theory.
- The delay monad is the free completely iterative monad over the identity functor.
- In general, the free completely iterative monad over a functor H is $A \mapsto \nu X.A + HX$.
- Complete iterativeness: Unique existence of a combinator satisfying the equation of repeat.
- Freeness: the "smallest" such monad.
- In a good mathematical sense, the delay monad is the universal one among the monads suitable for capturing iteration.

Constructive domain theory

- We have looping, what about recursion?
- Domains (posets with a bottom and lubs of all omega-chains):

```
class Dom a where
    bot :: a
    lub :: [a] -> a
```

- Some constructions of domains:

```
instance Dom b => Dom (a -> b) where
    bot a = bot
    lub fs a = lub (map (\ f -> f a) fs)

instance (Dom a, Dom b) => Dom (a, b) where
    bot = (bot, bot)
    lub abs = (lub (map fst abs), lub (map snd abs))
```

- Least fixpoints construction:

```
iterate :: (a -> a) -> a -> [a]
iterate f a = a : iterate f (f a)

lfp :: Dom a => (a -> a) -> a
lfp f = lub (iterate f bot)
```

- Delay types are domains:

```
instance Dom (Delay a) where
    bot = never
    lub = omegarace
```

- Partial ordering: $d \sqsubseteq d'$ iff $d \downarrow a$ implies $d' \downarrow a$ where $\downarrow$ is defined inductively by
  - $\mathsf{now}(a) \downarrow a$,
  - if $d \downarrow a$, then $\mathsf{later}(d) \downarrow a$.
- This is not antisymmetric, we only have a preordered set and to get a partial order, we must quotient wrt the symmetric closure.

- An example:

```
fib :: Integer -> Delay Integer
fib = lfp (\ fib -> \ n ->
              if n == 0 then return 0
              else if n == 1 then return 1
              else do x <- fib (n-1)
                      y <- fib (n-2)
                      return (x+y)
           )
```

A monadic interpreter

- A typed cbv language with integers and booleans.
- Term syntax:

```
type Var = String
data Tm = V Var | L Var Tm | Tm :@ Tm
        | Tm :& Tm | Fst Tm | Snd Tm
        | N Integer | Tm :+ Tm | ...
        | TT | FF | If Tm Tm Tm | ...
        -- looping
        | While Tm Tm | Until Tm Tm
        -- general recursion
        | Rec Var Tm
```

- Semantic domains:

```
data Val = I Integer | B Bool | P (Val, Val) | F (Val -> Delay Val)
type Env = [(Var, Val)]
```

- Evaluation:

```
ev :: Tm -> Env -> Delay Val
ev (V x)      env = return (unsafelookup x env)
ev (L x e)    env = return (F (\ a -> ev e (update x a env)))
ev (e :@ e') env = do F k <- ev e  env
                      a   <- ev e' env
                      k a
...
ev (While e e') env
     = do F p <- ev e  env
          F k <- ev e' env
          return (F (while'  (\ a -> do { B b <- g a ; return b } k)))
ev (Until e e') env
     = do F k <- ev e  env
          F p <- ev e' env
          return (F (repeat' k (\ a -> do { B b <- g a ; return b })))
ev (Rec x e)    env = return (F (lfp f))
     where f k a = do {F k' <- ev e (update x (F k) env) ; k' a }
```

- Example:

```
-- fib = rec fib -> \ n -> if n == 0 then 0
--                          else if n == 1 then 1
--                               else fib (n-1) + fib (n-2)

fib = Rec "fib" (L "n"
  (If (V "n" :== N 0) (N 0)
     (If (V "n" :== N 1) (N 1)
        ((V "fib" :@ (V "n" :- N 1)) :+ (V "fib" :@ (V "n" :- N 2))))))
```

Adding iteration to other monads

- For any monad, there is a monad supporting looping.

```
newtype Delay r a = D { unD :: r (Either a (Delay r a)) }
                                                        -- coinductive
instance Functor r => Functor (Delay r) where
  fmap f (D d) = D (fmap (either (Left . f) (Right . fmap f)) d)

instance Monad r => Monad (Delay r) where
  return a = D (return (Left a))
  D d >>= k = D (d >>= either (unD . k) (return . Right . (>>= k)))

repeat :: Monad r => (a -> Delay r (Either b a)) -> a -> Delay r b
repeat k a = k a >>= either return (D . return . Right . repeat k)
```

- The original monad can be embedded into the derived one.

```
lift :: Functor r => r a -> Delay r a
lift c = It (fmap Left c)
```

- In a more concise notation, instead of the monad $A \mapsto \nu X.\ A + X$, we are now considering the monad $A \mapsto \nu X.\ R(A + X)$ induced by a monad R.

  Quite importantly, this is not the same as $A \mapsto \nu X.\ A + RX$, which is the free completely iterative monad on R as a functor.