Partiality is an Effect

Venanzio Capretta
Thorsten Altenkirch
Tarmo Uustalu

WG 2.8, Kalvi, 1–4 Oct. 2005
The problem of partiality
The delay datatype and monad
Constructive domain theory and recursion
Combining partiality with other effects
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:
  
  \[
  \text{data Delay } a = \text{Now } a \mid \text{Later } (\text{Delay } a) \quad -- \text{coinductive}
  \]

- Infinite delay:
  
  \[
  \text{never :: Delay } a \\
  \text{never } = \text{Later } \text{never}
  \]

- Minimalization:
  
  \[
  \text{minim :: (Int } \to \text{ Bool}) \to \text{Delay } \text{Int} \\
  \text{minim } p = \text{minimFrom } p \ 0
  \]

  \[
  \text{minim :: (Int } \to \text{ Bool}) \to \text{Int } \to \text{Delay } \text{Int} \\
  \text{minimFrom } p \ n = \text{if } p \ n \text{ then Now } n \text{ else Later } (\text{minimFrom } p \ (n+1))
  \]
• Competition of two computations:
  \[
  \text{race} :: \text{Delay} \ a \rightarrow \text{Delay} \ a \rightarrow \text{Delay} \ a \\
  \text{race} (\text{Now} \ a) \_ = \text{Now} \ a \\
  \text{race} (\text{Later} \ _) (\text{Now} \ a) = \text{Now} \ a \\
  \text{race} (\text{Later} \ d) (\text{Later} \ d') = \text{Later} (\text{race} \ d \ d')
  \]

• Competition of omega many computations:
  \[
  \text{omegarace} :: \left[\text{Delay} \ a\right] \rightarrow \text{Delay} \ a \\
  \text{omegarace} (d:ds) = \text{race} \ d \ (\text{Later} \ (\text{omegarace} \ ds))
  \]

• Note that all function definitions above are guarded corecursive.
Monadic structure of Delay

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

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

- A special operation:

  ```haskell
  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:

\[
\text{while} :: (a \to \text{Delay (Either b a)}) \to \\
\quad \text{Delay (Either b a)} \to \text{Delay b} \\
\text{while k (Now (Left b))} = \text{Now b} \\
\text{while k (Now (Right a))} = \text{Later (while k (k a))} \\
\text{while k (Later c)} = \text{Later (while k c)}
\]

\[
\text{repeat} :: (a \to \text{Delay (Either b a)}) \to a \to \text{Delay b} \\
\text{repeat k a} = \text{while k (Now (Right a))}
\]
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.
• 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
  
  – $\text{now}(a) \downarrow a$,

  – if $d \downarrow a$, then $\text{later}(d) \downarrow a$.

• 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:

\[ev :: Tm \to Env \to \text{Delay Val}\]

\[ev (V x) \text{ env} = \text{return (unsafelookup x env)}\]

\[ev (L x e) \text{ env} = \text{return (F (\ a \to ev e (update x a env)))}\]

\[ev (e :0 e') \text{ env} = \text{do F k <- ev e env}\]
\[a <- ev e' env\]
\[k a\]

... 

\[ev (\text{While e e'}) \text{ env}\]
\[= \text{do F p <- ev e env}\]
\[F k <- ev e' env\]
\[\text{return (F (while' (\ a \to \text{do \{ B b <- g a ; return b \} k))}})\]

\[ev (\text{Until e e'}) \text{ env}\]
\[= \text{do F k <- ev e env}\]
\[F p <- ev e' env\]
\[\text{return (F (repeat' k (\ a \to \text{do \{ B b <- g a ; return b \} k}))}})\]

\[ev (Rec x e) \text{ env} = \text{return (F (lfp f))}\]

\[\text{where f k a = do \{F k' <- ev e (update x (F k) env) ; k' a \}}\]
Example:

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

```lisp
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)))))
)```
• 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 $A \leftrightarrow \nu X. A + X$, we are now considering the monad $A \leftrightarrow \nu X. R(A + X)$ induced by a monad $R$. Quite importantly, this is not the same as $A \leftrightarrow \nu X. A + RX$, which is the free completely iterative monad on $R$ as a functor.