Partiality is an Effect

Venanzio Capretta
Thorsten Altenkirch
Tarmo Uustalu

Dependently Typed Programming, Dagstuhl, 12–17 Sept. 2004
Outline

- Is partiality pure or not?
  - Motivating example
- Partiality by finite failure vs non-termination
- Lie management
- Capretta’s (or Adámek et al.’s?) monad for iteration
- Recursion in Capretta’s monad
- A quotient
- Adding iteration to other effects
- Signals and comonads
Example: Toy languages

- We think this is a pure implementation of an imperative language:

  ```plaintext
  semS :: Stm -> State -> State
  semS (x ::= a) st = upd x (semA a st) st
  semS Skip st = st
  semS (s1 :\ s2) st = semS s2 (semS s1 st)
  semS (If b s1 s2) st = if semB b st then semS s1 st else semS s2 st
  semS (While b s) st = if semB b st then semS (While b s) (semS s st)
    else st
  ```

- This is a pure implementation as well:

  ```plaintext
  semS :: Stm -> State -> [State]
  semS (x ::= a) st = return (upd x (semA a st) st)
  semS Skip st = return st
  semS (s1 :\ s2) st = do st’ <- semS s1 st
    st’’ <- semS s2 st’
    return st’’
  semS (If b s1 s2) st = if semB b st then semS s1 st else semS s2 st
  semS (While b s) st = semS (If b (s :\ While b s) Skip) st
  semS (s1 :\ s2) st = semS s1 st ++ semS s2 st
  ```
Partiality is an effect

- This is pure too, they say:

  \[
  \text{semS :: Stm} \to \text{State} \to \text{IO State}
  \]
  \[
  \text{semS (} x \::= a) \text{ st} = \text{return (upd x (semA a st) st)}
  \]
  \[
  \text{semS Skip st} = \text{return st}
  \]
  \[
  \text{semS (} s1 :\backslash \text{} s2) \text{ st} = \text{do st' <- semS s1 st}
  \]
  \[
  \text{st'' <- semS s2 st'}
  \]
  \[
  \text{return st''}
  \]
  \[
  \text{semS (If } b \text{ s1 s2) st} = \text{if semB b st then semS s1 st else semS s2 st}
  \]
  \[
  \text{semS (While } b \text{ s) st} = \text{semS (If b (s :\backslash \text{} While b s) Skip) st}
  \]
  \[
  \text{semS (Print a) st} = \text{do \{}
  \text{print (semA a st); return st } \}
  \]

- And everybody says this is not:

  \[
  \text{unsafePerformIO :: IO a} \to a
  \]
  \[
  \text{semS :: Stm} \to \text{State} \to \text{State}
  \]
  \[
  \text{semS (} x \::= a) \text{ st} = \text{upd x (semA a st) st}
  \]
  \[
  \text{semS Skip st} = \text{st}
  \]
  \[
  \text{semS (} s1 :\backslash \text{} s2) \text{ st} = \text{semS s2 (semS s1 st)}
  \]
  \[
  \text{semS (If } b \text{ s1 s2) st} = \text{if semB b st then semS s1 st else semS s2 st}
  \]
  \[
  \text{semS (While } b \text{ s) st} = \text{semS (If b (s :\backslash \text{} While b s) Skip) st}
  \]
  \[
  \text{semS (Print a) st} = \text{case unsafePerformIO (print (semA a st)) of}
  \]
  \[
  () \to \text{st}
  \]
• This biased and unfair. Because of While we can loop! Loops may not terminate. What’s pure about non-termination?

So our real situation is:

```haskell
unsafeRepeat :: (a -> Either b a) -> a -> b
unsafeRepeat f a = case f a of
  Left b -> b
  Right a' -> unsafeRepeat f a'
```

```haskell
semS :: Stm -> State -> State
semS (x ::= a) st = upd x (semA a st) st
semS Skip st = st
semS (s1 :\ s2) st = semS s2 (semS s1 st)
semS (If b s1 s2) st = if semB b st then semS s1 st else semS s2 st
semS (While b s) st = if semB b st then semS unsafeRepeat k st else st
  where k st = let st' = semS s st
              in case semB b st' of
                   True -> Right st'
                   False -> Left st'
```

Partiality is an effect
Problem

- We have an impure combinator

  \[
  \text{unsafeRepeat} :: (a \rightarrow \text{Either } b \text{ a}) \rightarrow a \rightarrow b
  \]

  \[
  \text{unsafeRepeat } f \ a = \text{case } f \ a \text{ of }
  \]

  \[
  \text{Left } b \rightarrow b
  \]

  \[
  \text{Right } a' \rightarrow \text{unsafeRepeat } f \ a'
  \]

- We would like to have a pure combinator

  \[
  \text{repeat} :: (a \rightarrow M (\text{Either } b \text{ a})) \rightarrow a \rightarrow M b
  \]

  for some monad encapsulating the impurity.
Partiality is an effect

Finite failure vs. non-termination

- The error monad is perfect for partiality from finite failure (e.g., because of a pattern-match failure), but useless for non-termination.

- Or shall we try?

```haskell
data Eugenio a = Eventually a | Never

RepeatE0 :: (a -> Eugenio (Either b a)) -> a -> Eugenio b
RepeatE0 f a = case f a of
    Eventually (Left b) -> Eventually b
    Eventually (Right a') -> RepeatE0 f a'
    Never -> Never
```

This has not captured the possibility for non-termination.
- How do we improve??

```
repeatE :: (a -> Eugenio (Either b a)) -> a -> Eugenio b
repeatE f a = let
    c = RepeatE0 f a
    in if halting c then c else Never
```

This can’t be serious!...

- More seriously, instead of the error monad one needs a “lifting” monad...
Conor’s story from Monday

- Idea: Use the reader monad and you can be clean... until you consult your environment.

- The general parameterized reader monad:

  newtype Reader r a = Reader { runReader :: r -> a }

  instance Functor (Reader r) where
  fmap f c = Reader (\ r -> f (runReader c r))

  instance Monad (Reader r) where
  return a = Reader (\ _ -> a)
  c >>= k = Reader (\ r -> runReader (k (runReader c r)) r)
The specific instance for iteration-like combinators from the environment:

\[
\text{newtype } \text{URT} = \text{URT} \{\ \text{unURT} :: \text{forall} \ a \ b. \\
(a \to \text{Either} \ b \ a) \to a \to b \}
\]

type Conor \ a = \text{Reader} \ \text{URT} \ a

\[
\text{repeatC} :: (a \to \text{Conor} \ (\text{Either} \ b \ a)) \to a \to \text{Conor} \ b \\
\text{repeatC} \ k \ a = \text{Reader} (\ \lambda \ ur \to \\
\ \ \ \ \ \ \ \ \ \ \text{unURT} \ ur (\ \lambda \ a' \to \text{runReader} \ (k \ a') \ ur) \ a)
\]

Your environment tells you a big lie:

\[
\text{trustBigLie} :: \text{Conor} \ a \to a \\
\text{trustBigLie} \ c = \text{runReader} \ c \ (\text{URT} \ \text{unsafeRepeat})
\]
Capretta’s monad

- Idea: Take waiting seriously, charge a unit cost for every iteration cycle.
- The parameterized delay datatype:

  data Venanzio a = Now a | Later (Venanzio a) -- coinductive

  outV :: Venanzio a -> Either a (Venanzio a)
  outV (Now a) = Left a
  outV (Later c) = Right c
- It’s ok to use coiteration and primitive corecursion:

  unfoldV :: (x -> Either a x) -> x -> Venanzio a
  unfoldV p x = case p x of
      Left  a  -> Now a
      Right x' -> Later (unfoldV p x')

  corecV :: (x -> Either a (Either (Venanzio a) x)) -> x -> Venanzio a
  corecV p x = case p x of
      Left  a  -> Now a
      Right (Left c) -> Later c
      Right (Right x') -> Later (corecV p x')

- Or one may use general guarded-by-constructions corecursion.
Partiality is an effect

- First example: infinite waiting:

  never :: Venanzio a

  never = Later never

  -- coiterative

- The delay monad:

  instance Functor Venanzio where

  fmap f (Now a) = Now (f a)

  fmap f (Later c) = Later (fmap f c)

  -- coiterative

  instance Monad Venanzio where

  return a = Now a

  Now a >>= k = k a

  -- primitive corecursive

  Later c >>= k = Later (c >>= k)
- Iteration (not obviously structurally corecurcive):

\[
\text{repeatV :: } (a -> \text{Venanzio (Either b a)}) -> a -> \text{Venanzio b}
\]

\[
\text{repeatV f a = f a >>= \ c -> case c of}
\]

\[
\text{Left b -> Now b}
\]

\[
\text{Right a' -> Later (repeatV f a')}
\]

- An alternative definition (obviously coiterative, but non-trivially equivalent to the previous one):

\[
\text{repeatV :: } (a -> \text{Venanzio (Either b a)}) -> a -> \text{Venanzio b}
\]

\[
\text{repeatV f a = whileV f (Now (Right a))}
\]

\[
\text{whileV :: } (a -> \text{Venanzio (Either b a)}) ->
\]

\[
\text{Venanzio (Either b a) -> Venanzio b}
\]

\[
\text{whileV f (Now (Left b)) = Now b}
\]

\[
\text{whileV f (Now (Right a)) = Later (whileV f (f a))}
\]

\[
\text{whileV f (Later c) = Later (whileV f c)}
\]
Capretta vs Adámek, Milius et al.

- The Capretta monad $A \mapsto \nu X.A + X$ has been discussed extensively in category theory.
- It 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, Capretta’s monad is the universal one among the monads suitable for capturing iteration.
Recursion in Capretta’s monad

- Idea: Arrange for a race between all finite approximations of the fixedpoint.

\[
\text{generalV} :: ((a \to \text{Venanzio } b) \to a \to \text{Venanzio } b) \to a \to \text{Venanzio } b
\]

\[
\text{generalV \ phi \ a} = \text{aux} (\_ \to \text{never})
\]

where \[
\text{aux} \ k = \text{race} \ (k \ a) \ (\text{Later} \ (\text{aux} \ (\phi \ k)))
\]

\[
\text{race} :: \text{Venanzio } b \to \text{Venanzio } b \to \text{Venanzio } b
\]

\[
\text{race} \ (\text{Now} \ b0) \ _ \ = \ \text{Now} \ b0
\]

\[
\text{race} \ (\text{Later} \ _) \ (\text{Now} \ b1) = \ \text{Now} \ b1
\]

\[
\text{race} \ (\text{Later} \ c0) \ (\text{Later} \ c1) = \ \text{Later} \ (\text{race} \ c0 \ c1)
\]
A quotient

- Idea: Forget about the cost bookkeeping and get half-way back to the error monad.
- First identify all bisimilar elements in the coinductive type.
- Define an equivalence relation \( \sim \) inductively by
  \[
  \frac{c \sim c}{c} \quad \frac{c \sim d \text{ later } c}{c \sim d} \quad \frac{c \sim d \text{ later } d}{c \sim d}
  \]
- Define also a consistency relation \( \land \) coinductively by the rules
  \[
  \frac{c \land c}{c \land c} \quad \frac{c \land d \text{ later } c \land d}{c \land d} \quad \frac{c \land d \text{ later } d}{c \land d}
  \]

This is not transitive, but it is closed under \( \sim \).
- Now the quotient of Capretta’s monad wrt. \( \sim \) is a monad too.

- One has to verify that all operations are still well-defined.

  For \texttt{race}, one has to require that the two arguments are consistent.

  This is met in the calls to \texttt{race} in \texttt{generalV}. 

}\end{slide}
Adding iteration to other monads

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

```haskell
newtype Iter r a = It { unIt :: r (Either a (Iter r a)) }
                   -- coinductive

instance Functor r => Functor (Iter r) where
    fmap f (It c) = It (fmap (either (Left . f) (Right . fmap f)) c)

instance Monad r => Monad (Iter r) where
    return a = It (return (Left a))
    It c >>= k = It (c >>= either (unIt . k) (return . Right . (>>= k)))

mrepeate :: Monad r => (a -> Iter r (Either b a)) -> a -> Iter r b
mrepeate f a = f a >>= either return (It . return . Right . repeatI f)

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

    lift :: Functor r => r a -> Iter 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.