Advanced Functional Programming

Continuations

- Continuation passing style
- Continuation monad
- Throw and catch
- Callcc

Continuations

For any function f, of type

```
f :: a -> b -> c
```

Its continuation style is

```
f :: a -> b -> (c -> ans) -> ans
```

This allows the user to control the flow of control in the program. A program in continuation passing style (CPS) has all functions in this style.

e.g. `(+) :: Int -> Int -> (Int -> ans) -> ans`

Lists in CPS

```
-- old (direct) style
append [] xs = xs
append (y:ys) xs = y : (append ys xs)

-- CPS style
consC :: a -> [a] -> ([a] -> ans) -> ans
consC x xs k = k(x:xs)

appendC :: [a] -> [a] -> ([a] -> ans) -> ans
appendC [] xs k = k xs
appendC (y:ys) xs k =
    appendC ys xs (\ zs -> consC y zs k)
```

Flattening Trees in CPS

```
data Tree a = Tip a | Fork (Tree a) (Tree a)

-- direct style
flat :: Tree a -> [a]
flat (Tip x) = x : []
flat (Fork x y) = flat x ++ flat y

-- CPS style
flatC :: Tree a -> ([a] -> ans) -> ans
flatC (Tip x) k = consC x [] k
flatC (Fork x y) k =
    flatC y (\ zs ->
    flatC x (\ ws -> appendC ws zs k))
```

Remember this pattern

What's this good for?

Is it efficient?

```
tree1 = Fork (Fork (Tip 1) (Tip 2))
             (Fork (Tip 3) (Tip 4))

double 0 x = x
double n x = double (n-1) (Fork x x)
```

Try both versions on some big trees

How many nodes in this tree

```
ex1 = length(flat (double 14 tree1))
ex2 = length(flatC (double 14 tree1) id)
```

Test results

```
Main> :set +s
Main> ex1
65536
(1179828 reductions, 2359677 cells, 10 garbage collections)
Main> ex2
65536
(2425002 reductions, 5505325 cells, 34 garbage collections)
```

Clearly the continuation example uses more resources!

Why use it?

Advantages of CPS

Use continuations for explicit control of control flow

Consider a function

```
prefix :: (a -> Bool) -> [a] -> Maybe[a]
```

`(prefix p xs)` returns the longest prefix of xs, ys such that

```
(all p ys) &&
not(p (head (drop (length ys) xs)))
```

I.e. the next element does not have the property p. Return nothing if all elements meet p.

```
ex3 = prefix even [2,4,6,5,2,4,8]

Main> ex3
Just [2,4,6]

ex4 = prefix even [2,4,6,8,10,12,14]

Main> ex4
Nothing
```

Code

```
prefix :: (a -> Bool) -> [a] -> Maybe [a]
prefix p [] = Nothing
prefix p (x:xs) = if p x
                     then cons x (prefix p xs)
                     else Just []
  where cons x Nothing = Nothing
        cons x (Just xs) = Just(x:xs)
```

- What happens if everything in the list meets p?
- How many calls to cons?
- Can we do better? Use continuations!

Prefix in CPS

```
prefixC :: (a -> Bool) -> [a] ->
           (Maybe [a] -> Maybe ans) -> Maybe ans
```

Note the discarded continuation!

prefixC is tail recursive!

```
prefixC p [] k = Nothing
prefixC p (x:xs) k =
    if p x
       then prefixC p xs (cons x k)
       else k (Just [])
  where cons x k (Just xs) = k (Just(x:xs))
        cons x k Nothing =
                 error "This case is never called“
```

How many times is cons called if p is never false?

The continuation denotes normal control flow, by never using it we can short circuit the normal flow!

Style

```
prefixC p [] k = Nothing
prefixC p (x:xs) k =
    if p x
       then prefixC p xs (cons x k)
       else k (Just [])
  where cons x k (Just xs) = k (Just(x:xs))
        cons x k Nothing =
                 error "This case is never called“

prefixC p [] k = Nothing
prefixC p (x:xs) k =
    if p x
       then prefixC p xs (\ (Just xs) ->
            k(Just(x:xs)))
       else k (Just [])
```

The continuation monad

```
data Cont ans x = Cont ((x -> ans) -> ans)
runCont (Cont f) = f

instance Monad (Cont ans) where
  return x = Cont ( \ f -> f x )
  (Cont f) >>= g =
     Cont( \ k -> f (\ a -> runCont (g a)
                         (\ b -> k b)) )

throw :: a -> Cont a a
throw x = Cont(\ k -> x)

force :: Cont a a -> a
force (Cont f) = f id
```

Prfefix in Monadic style

```
prefixK :: (a -> Bool) -> [a] -> Cont (Maybe[a]) (Maybe[a])

    prefixK p [] = throw Nothing
    prefixK p (x:xs) =
        if p x then do { Just xs <- prefixK p xs
                       ; return(Just(x:xs)) }
               else return(Just [])
```

- Note how throw is a global abort.
- Its use is appropriate whenever local failure, implies global failure.

Pattern Matching

```
data Term = Int Int | Pair Term Term

data Pat = Pint Int
         | Ppair Pat Pat
         | Pvar String
         | Por Pat Pat

type Sub = Maybe[(String,Term)]

instance Show Term where
  show (Int n) = show n
  show (Pair x y) =
        "("++show x++","++show y++")"
```

Match function

```
match :: Pat -> Term -> Sub

match (Pint n) (Int m) =
   if n==m then Just[] else Nothing
match (Ppair p q) (Pair x y) =
   match p x .&. match q y
match (Pvar s) x = Just[(s,x)]
match (Por p q) x = match p x .|. match q x
match p t = Nothing
```

Example tests

```
t1 = Pair (Pair (Int 5) (Int 6)) (Int 7)
p1 = Ppair (Pvar "x") (Pvar "y")
p2 = Ppair p1 (Pint 1)
p3 = Ppair p1 (Pint 7)
p4 = Por p2 p3


Main> match p1 t1
Just [("x",(5,6)),("y",7)]
Main> match p2 t1
Nothing
Main> match p3 t1
Just [("x",5),("y",6)]
Main> match p4 t1
Just [("x",5),("y",6)]
```

Match in CPS

```
matchC :: Pat -> Term -> (Sub -> Maybe ans) -> Maybe ans
matchC (Pint n) (Int m) k =
   if n==m then k(Just[]) else Nothing
matchC (Ppair p q) (Pair x y) k =
    matchC p x (\ xs ->
    matchC q y (\ ys ->
    k(xs .&. ys)))
matchC (Pvar s) x k = k(Just[(s,x)])
matchC (Por p q) x k =
    matchC p x (\ xs ->
    matchC q x (\ ys ->
    k(xs .|. ys)))
```

Note the discarded continuation!

- Why does this return nothing?

```
ex8 = matchC p4 t1 id
Main> ex8
Nothing
```

Two continuations

- Here is an example with 2 continuations
- A success continuation, and a failure continuation

```
matchC2 :: Pat -> Term -> (Sub -> Sub) -> (Sub -> Sub) ->
   Sub
matchC2 (Pint n) (Int m) good bad =
   if n==m then good(Just[]) else bad Nothing
matchC2 (Ppair p q) (Pair x y) good bad =
    matchC2 p x (\ xs ->
    matchC2 q y (\ ys ->
    good(xs .&. ys)) bad) bad
matchC2 (Pvar s) x good bad = good(Just[(s,x)])
matchC2 (Por p q) x good bad =
    matchC2 p x good (\ xs ->
    matchC2 q x good bad)
matchC2 _ _ good bad = bad Nothing
```

Tests

```
t1 = Pair (Pair (Int 5) (Int 6)) (Int 7)
p1 = Ppair (Pvar "x") (Pvar "y")
p2 = Ppair p1 (Pint 1)
p3 = Ppair p1 (Pint 7)
p4 = Por p2 p3

ex9 = matchC2 p4 t1 id id

Main> ex10
Just [("x",5),("y",6)]
```

Fixing matchC

```
matchK :: Pat -> Term -> (Sub -> Maybe ans) -> Maybe ans

matchK (Pint n) (Int m) k =
   if n==m then k(Just[]) else Nothing
matchK (Ppair p q) (Pair x y) k =
    matchK p x (\ xs ->
    matchK q y (\ ys ->
    k(xs .&. ys)))
matchK (Pvar s) x k = k(Just[(s,x)])
matchK (Por p q) x k =
    case matchK p x id of
       Nothing -> matchK q x k
       other -> k other
```

Note the intermediate id continuation

Not the ultimate use of the original continuation

- Note the pattern here of "catching" a possible local failure, and then picking up where that left off

Catch and Throw

```
throw :: a -> Cont a a
throw x = Cont(\ k -> x)

catch :: Cont a a -> Cont b a
catch (Cont f) = Cont g
  where g k = k(f id)
```

- Throw causes the current computation to be abandonned. **`(catch x)`** runs **`x`** in a new continuation and then applies the continuation to the result.
- **`(catch x) == x`** when **`x`** does not throw.

Match in monadic style

```
matchK2 :: Pat -> Term -> Cont Sub Sub

matchK2 (Pint n) (Int m) =
  if n==m then return(Just[])
          else throw Nothing
matchK2 (Ppair p q) (Pair x y) =
  do { a <- matchK2 p x
     ; b <- matchK2 q y
     ; return(a .&. b) }
matchK2 (Pvar s) x = return(Just[(s,x)])
matchK2 (Por p q) x =
  do { a <- catch(matchK2 p x)
     ; case a of
         Nothing -> matchK2 q x
         other -> return other
     }
```

Interpreters in CPS

```
data Exp = Var String
         | Lam String Exp
         | App Exp Exp
         | Num Int
         | Op (Int -> Int -> Int) Exp Exp

data V = Fun (V -> (V -> V) -> V)
       | N Int

plus,times,minus :: Exp -> Exp -> Exp
plus x y = Op (+) x y
times x y = Op (*) x y
minus x y = Op (-) x y

extend :: Eq a => (a -> b) -> b -> a -> a -> b
extend env v a b = if a==b then v else env b
```

Eval in CPS

```
eval :: (String -> V) -> Exp -> (V -> V) -> V
eval env (Var s) k = k(env s)
eval env (App x y) k =
   eval env x (\ (Fun f) ->
   eval env y (\ z ->
   f z k))
eval env (Lam s x) k =
   k(Fun (\ v k2 -> eval (extend env v s) x k2))
eval env (Num n) k = k(N n)
eval env (Op f x y) k =
   eval env x (\ (N a) ->
   eval env y (\ (N b) ->
   k (N(f a b))))
```

Eval in monadic style

Note that the value datatype (U) must be expressed using the monad

```
type C x = Cont U x
data U = Fun2 (U -> C U)
       | N2 Int

eval2 :: (String -> U) -> Exp -> C U
eval2 env (Var s) = return(env s)
eval2 env (App f x) =
   do { Fun2 g <- eval2 env x
      ; y <- eval2 env x
      ; g y }
eval2 env (Lam s x) =
   return(Fun2(\ v -> eval2 (extend env v s) x))
eval2 env (Op f x y) =
   do { N2 a <- eval2 env x
      ; N2 b <- eval2 env y
      ; return(N2(f a b)) }
eval2 env (Num n) = return(N2 n)
```

CPS is good when the language has fancy control structures

```
data Exp = Var String
         | Lam String Exp
         | App Exp Exp
         | Num Int
         | Op (Int -> Int -> Int) Exp Exp
         | Raise Exp
         | Handle Exp Exp

type C3 x = Cont W x
data W = Fun3 (W -> C3 W)
       | N3 Int
       | Err W
```

```
eval3 :: (String -> W) -> Exp -> C3 W
eval3 env (Var s) = return(env s)
eval3 env (App f x) =
   do { Fun3 g <- eval3 env x
      ; y <- eval3 env x; g y }
eval3 env (Lam s x) =
   return(Fun3(\ v -> eval3 (extend env v s) x))
eval3 env (Op f x y) =
   do { N3 a <- eval3 env x
      ; N3 b <- eval3 env y
      ; return(N3(f a b)) }
eval3 env (Num n) = return(N3 n)
eval3 env (Raise e) =
   do { x <- eval3 env e; throw(Err x) }
eval3 env (Handle x y) =
   do { x <- catch (eval3 env x)
      ; case x of
         Err v -> do { Fun3 g <- eval3 env y; g v }
         v -> return v
      }
```