Continuations

The rest of the computation.

\[ 1 + (\lambda x \to x \times 3) \ (1 + 1) \]

\[ (+) \ 1 \ ((\lambda x \to (*) \ x \ 3) \ ((+) \ 1 \ 1)) \]

Call-by-value breakdown:

\[
\begin{align*}
\text{let } v1 &= (+); v2 = 1 \\
& \quad v3 = (\lambda v \to \text{let } v1 = (*); v2 = 3 \ \text{in } v1 \ v \ v3) \\
& \quad v4 = (+); v5 = 1; v6 = 1 \\
& \quad v7 = v4 \ v5 \ v6 \\
& \quad v8 = v3 \ v7 \\
\text{in } v1 \ v2 \ v8
\end{align*}
\]
Continuation Passing Style

Making continuations explicit.

```
plus e1 e2 k = e1 $ \v1 ->
    e2 $ \v2 ->
    k $ v1 + v2

(^) x k = k x
```

```
plus (plus (1 ^) (2 ^)) (plus (3 ^) (4 ^)) id ⟷ 10
```

CPS expressions require an initial continuation to know what to do with the answer.

Any term can be transformed into CPS.
- Continuations and CPS studied since mid 60’s.

- CPS makes control flow explicit.

- CPS used to structure compilers.

- CPS is very similar to Monadic style.
CPS Terms & Values

type K v = (v -> Answer) -> Answer

data Term = Var String
    | String :\ Term
    | Term :$: Term

data Value = Err String
    | Fun (Value -> K Value)
**CPS Interpreter**

type Answer = Value

ggetVar :: Env -> String -> K Value
ggetVar ((x,v):xs) y k = if x == y then k v else getVar xs y k
ggetVar [] y k = k $ Err $ "unbound variable "++y

apply :: Value -> Value -> K Value
apply (Fun v1) v2 k = v1 v2 k
apply f _ k = k $ Err $ "expected function, found "++(show f)

interp :: Term -> Env -> K Value
interp (Var x) env k = getVar env x k
interp (x :\ e) env k = k $ Fun $ \v -> interp e ((x,v):env)
interp (e1 :$: e2) env k =
    interp e1 env $ \v1 ->
    interp e2 env $ \v2 ->
    apply v1 v2 k

test e = interp e [] id
The Current Continuation

data Term = ... | CallCC String Term

callcc :: ((v -> K w) -> K v) -> K v
callcc f k = f (\a k’ -> k a) k

interp (CallCC x e) env k =
    callcc (\v -> interp e ((x,Fun v):env)) k

Note:

callcc :: ((v -> (w -> Answer) -> Answer) -> Answer) -> (v -> Answer) -> Answer) ->
        (v -> Answer)
        -> Answer
Using CallCC

Access to the current continuation opens up many possibilities.

\[
\text{Con 1} ::= \text{CallCC "k" (Con 2) ::= (Var "k" :: (Var "k" :: Con 3))}
\]

\[
\text{Con 1} ::= \text{CallCC "k" (Con 2) ::= (Var "k" :: (Var "k" :: Con 3))}
\]

\[
\text{Con 1} ::= \text{CallCC "k" (Con 2) ::= Con 3}
\]

\[
\text{CallCC "k" $ "x" :: (Var "k") :: ("y" :: Var "x" :: Var "y")}
\]

Can use CallCC to get static exceptions:

\[
\text{handle e e' = CallCC "k" (e :: (_ :: Var "k" :: e'))}
\]

\[
\text{raise e = e :: Skip}
\]

\[
\text{"x" :: handle ("ex" :: If (Var "x" == Con 10) (raise (Var "ex")) (Var "x" :: Con 2) (Con (-1)))}
\]

Many other strange and complex possibilities.
Extending CPS Interpreter

data Term = ... | Skip | Out

data Value = ... | Unit

type Answer = (String, Value)

out :: Show a => a -> K Value
out v k = let (o, v1) = k Unit in ((show v)++"; "++o, v1)

interp Skip env k = k Unit
interp (Out e) env k = interp e env $ \v -> out v k

Other extensions similarly done.
Monads via CPS

type Answer = M Value

type M a = Int \rightarrow (String, (a, Int))

tick :: K ()
tick k = bind ($s \rightarrow (\"\",((),s+1))) k

out :: Show a \Rightarrow a \rightarrow K Value
out v k = bind ($s \rightarrow ((show v)++; \\,,(Unit, s))) k

apply :: Value \rightarrow Value \rightarrow K Value
apply (Fun v1) v2 k = tick $ \_ \rightarrow v1 v2 k
apply f _ k = k $ Err $ "expected function, found \\"++(show f)
**CPS via Monads**

\[\text{type } K \ a = (a \to \text{Answer}) \to \text{Answer}\]

\[\text{unit :: } a \to K\ a\]
\[\text{unit } a = \k \to k \ a\]

\[\text{bind :: } K\ a \to (a \to K\ b) \to K\ b\]
\[\text{bind } m\ f = \k \to m (\v \to f \ v \ k)\]

\[\text{bind unit } x = x\]

\[\x \to \text{bind } f (\text{unit } x) = f\]

\[\x \to \text{bind } (\y \to \text{bind } g (f\ y))\ x = \x \to \text{bind } g (\text{bind } f\ x)\]
Conclusion

- Can write monads as continuations.

- Can write continuations as monads.

- Slightly easier to abstract monadic types—have to include special application functions to use abstract CPS types.