Continuations

Dr. Mattox Beckman

Illinois Institute of Technology
Department of Computer Science

Objectives

You should be able to...

It is possible to use functions to represent the *control flow* of a program. This technique is called *continuation passing style*. After today's lecture, you should be able to

- explain what CPS is,
- give an example of a programming technique using CPS, and
- transform a simple function from direct style to CPS.

Direct Style

Example Code

```haskell
inc x = x + 1
double x = x * 2
half x = x `div` 2

result = inc (double (half 10))
```

- Consider the function call above. What is happening?

The Continuation

```
result = inc (double (half 10))
```

- We can 'punch out' a subexpression to create an expression with a 'hole' in it.

  `result = inc (double [[]])`
- This is called a *context*. After `half 10` runs, its result will be put into this context.
- We can call this context a *continuation*.

Making Continuations Explicit

- We can make continuations explicit in our code.

```
cont = \ v -> inc (double v)
```

- Instead of returning, a function can take a *continuation argument.*

Using a Continuation

```
half x k = k (x `div` 2)
result = half 10 cont
```

- Convince yourself that this does the same thing as the original code.

Properties of CPS

- A function is in *Direct Style* when it returns its result back to the caller.
- A *Tail Call* occurs when a function returns the result of another function call without processing it first.
    - This is what is used in accumulator recursion.
- A function is in *Continuation Passing Style* when it passes its result to another function.
    - Instead of returning the result to the caller, we pass it forward to another function.
    - Functions in CPS “never return”.
- Lets see some more examples.

Comparisons

Direct Style

```haskell
inc x = x + 1
double x = x * 2
half x = x `div` 2

result = inc (double (half 10))
```

CPS

```haskell
inc x k = k $ x + 1
double x k = k $ x * 2
half x k = k $ x `div` 2
id x = x
result = half 10 (\v1 ->
           double v1 (\v2 ->
           inc v2 id))
```

CPS and Imperative style

- CPS look like imperative style if you do it right.

CPS

```
result = half 10 (\v1 ->
            double v1 (\v2 ->
            inc v2 id))
```

Imperative Style

```
    v1  := half 10
    v2  := double v1
result  := inc v2
```

The GCD Program

```
gcd a b | b == 0 = a
        | a < b  = gcd b a
        | otherwise = gcd b (a `mod` b)
```

`gcd 44 12` ⇒ `gcd 12 8` ⇒ `gcd 8 4` ⇒ `gcd 4 0` ⇒ 4

- The running time of this function is roughly $\mathcal{O}(lg\ a)$.

GCD of a list

```
gcdstar [] = 0
gcdstar (x:xs) = gcd x (gcdstar xx)

> gcdstar [44, 12, 80, 6]
2
> gcdstar [44, 12]
4
```

- Question: What will happen if there is a 1 near the beginning of the sequence?

Bad Solution I — Check and Return

```
bad1 [] = 0
bad1 (1:xx) = 1
bad1 (x:xs) = gcd x (gcdstar xx)
```

- This stops the computation, but a lot of work has already been done.

Bad Solution II — Goto Statement

```
1 bad2 [] = 0
2 bad2 (1:xx) = goto 4
3 bad2 (x:xs) = gcd x (gcdstar xx)
4 return 1
```

- Of course, this is nonsense.

Okay Solution – Prefiltering

```
gcdstar3 xx =
   if (all (\x -> x != 1) xx)
      then gcdstar xx
      else 1
```

- Of course, this would be a short lecture if we were content with that.

Definition of a Continuation

- A *continuation* is a function into which is passed the result of the current function's computation.

```
> report x = x
> plus a b k = k (a + b)
> plus 20 33 report
53
> plus 20 30 (\x-> plus 5 x report)
55
```

Continuation Solution

```
gcdstar xx k = aux xx k
  where aux [] newk = newk 0
        aux (1:xs) newk = k 1
        aux (x:xs) newk = aux xs (\res -> newk (gcd x res))

> gcdstar [44, 12, 80, 6] report
2
> gcdstar [44, 12, 1, 80, 6] report
1
```

More Vocab!

Tail Position A subexpression *s* of expressions *e*, if it is evaluated, will be taken as the value of *e*.

- `if x> 3 then x + 2 else x - 4`
- `let x = 5 in x + 4`
- `f (x * 3)` — no tail position here.

Tail Call A function call that occurs in tail position.

- `if (h x) then h x else x + (g x)`

Available A function call that can be executed by the current expression. The fastest way to be unavailable is to be guarded by an abstraction (anonymous function).

- `if h x then f x else x + g x`
- `if h x then (λ x -> f x) else x + g x`

Find the Tail Calls!

What expressions are in tail position? What expressions are tail calls? What calls are available?

```
foo [] = b
foo (0:xs) = foo xs
foo (1:xs) = let y = \ yy -> foo yy
                in foo xs
foo (x:xs) = z + foo xs
```

The CPS Transform, Steps 1 and 2

Step 1 Add a continuation argument to any function call

$$C[\![let\ f\ arg\ =\ e)]\!] \Rightarrow let\ f\ arg\ k\ =\ C[\![e]\!]$$

- The idea is that every function is going to take an extra parameter. "To whom should I tell the result?"

Step 2 A simple expression in tail position should be passed to a continuation instead of returned.

$$C[\![a]\!] \Rightarrow\ k\ a$$

assuming a is a constant or variable.

- "Simple" = "No available function calls."

The CPS Transform, Steps 3 and 4

Step 3 To a function call in tail position, pass the current continuation.

$$C[\![f\ arg]\!] \Rightarrow\ f\ arg\ k$$

- The function "isn't going to return," so we need to tell it where to put the result.

Step 4 A function call not in tail position needs to be built into a new continuation. Be sure your new continuation calls the old one if appropriate!

$$C[\![(op\ (f\ arg))]\!] \Rightarrow\ ((f\ arg)\ (\lambda\ r\ \rightarrow k(C[\![op]\!]\ r)))$$

Example

```haskell
foo [] = b
foo (0:xs) = foo xs
foo (x:xs) = x + foo xs
```

```haskell
foo [] k = k b
foo (0:xs) k = foo xs k
foo (x:xs) k = foo xs (\r -> k (x + r))
```

You try...

Do the map / foldr CPS activity.

Other Topics

- Continuations can simulate exceptions.
- They can also simulate cooperative multitasking.
  - These are called co-routintes.
- Some advanced routines are also available: call/cc, shift, reset.