Continuations

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
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

```
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 $\Rightarrow$ gcd 12 8 $\Rightarrow$ gcd 8 4 $\Rightarrow$ gcd 4 0 $\Rightarrow$ 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?
- We can use a continuation to handle this case.

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
```

The CPS Transform, Simple Expressions

Top Level Declaraion To convert a declaration, add a continuation argument to it and then convert the body.

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

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

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

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

The CPS Transform, Function Calls

Function Call on Simple Argument To a function call in tail position (where `arg` is simple), pass the current continuation.

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

Function Call on Non-simple Argument If `arg` is not simple, we need to convert it first.

$$C[\![f\ arg]\!]_k \Rightarrow\ C[\![arg]\!]_{(\lambda v.f\ v\ k)}, \text{ where } v \text{ is fresh.}$$

Example

```
foo 0 = 0
foo n | n < 0      = foo n
      | otherwise = inc (foo n)

foo 0 k = k 0
foo n k | n < 0      = foo n k
        | otherwise = foo n (\v -> inc v k)
```

The CPS Transform, Operators

Operator with Two Simple Arguments If both arguments are simple, then the whole thing is simple.

$$C[\![e_1 + e_2]\!]_k \Rightarrow \; k(e_1 + e_2)$$

Operator with One Simple Argument If e_2 is simple, we transform e_1.

$$C[\![e_1 + e_2]\!]_k \Rightarrow \; C[\![e_1]\!]_{(\lambda v -> k(v+e2))} \text{where } v \text{ is fresh.}$$

Operator with No Simple Arguments If both need to be transformed...

$$C[\![e_1 + e_2]\!]_k \Rightarrow \; C[\![e_1]\!]_{(\lambda v_1 -> C[\![e_2]\!]_{\lambda v_2 -> k(v_1+v_2)})} \text{where } v_1 \text{ and } v_2 \text{ ar}$$

Notice that we need to nest the continuations!

Examples

```
foo a b = a + b
bar a b = inc a + b
baz a b = a + inc b
quux a b = inc a + inc b

foo a b k = k a + b
bar a b k = inc a (\v -> k (v + b))
baz a b k = inc b (\v -> k (a + v))
quux a b k = inc a (\v1 -> inc b (\v2 -> k (v1 + v2)))
```

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.