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

```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

\[ \text{result} = \text{inc} \left( \text{double} \left( \text{half} \ 10 \right) \right) \]

- We can ‘punch out’ a subexpression to create an expression with a ‘hole’ in it.
  \[ \text{result} = \text{inc} \left( \text{double} \ [\ ] \right) \]
- This is called a context. After \text{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.
  \[
  \text{cont} = \lambda \, v \rightarrow \text{inc} \,(\text{double} \, v)
  \]
- Instead of returning, a function can take a continuation argument.

Using a Continuation

\[
\text{half} \, x \, k = k \,(x \ `\text{div}` \, 2)
\]
\[
\text{result} = \text{half} \, 10 \, \text{cont}
\]

- 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”.
- Let’s see some more examples.
Comparisons

Direct Style

\[
\begin{align*}
\text{inc } x &= x + 1 \\
\text{double } x &= x \times 2 \\
\text{half } x &= x \div 2 \\
\text{result } &= \text{inc } (\text{double } (\text{half } 10))
\end{align*}
\]

CPS

\[
\begin{align*}
\text{inc } x \ k &= k \ (x + 1) \\
\text{double } x \ k &= k \ (x \times 2) \\
\text{half } x \ k &= k \ (x \div 2) \\
\text{id } x &= x \\
\text{result } &= \text{half } 10 \ (\text{\v1 } \rightarrow \\
& \text{double } \text{\v1 } \ (\text{\v2 } \rightarrow \\
& \text{inc } \text{\v2 } \text{id}))
\end{align*}
\]
CPS and Imperative style

CPS

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

Imperative Style

```c
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 $O(lg \ a)$. 
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 \( 20 \ 30 \) (\(\lambda x \to\) 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 Declaration  To convert a declaration, add a continuation argument to it and then convert the body.

\[
C[f \text{ arg } = e]) \Rightarrow f \text{ arg } k = C[e]_k
\]

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 \( \text{arg} \) is simple), pass the current continuation.

\[
C[f \text{arg}]_k \Rightarrow f \text{arg} \ k
\]

Function Call on Non-simple Argument  If \( \text{arg} \) is not simple, we need to convert it first.

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

\[
\begin{align*}
\text{foo} \ 0 & = 0 \\
\text{foo} \ n \ | \ n < 0 & = \text{foo} \ n \\
& \quad | \ \text{otherwise} = \text{inc} \ (\text{foo} \ n) \\
\text{foo} \ 0 \ k & = k \ 0 \\
\text{foo} \ n \ k \ | \ n < 0 & = \text{foo} \ n \ k \\
& \quad | \ \text{otherwise} = \text{foo} \ n \ (\ \nu \rightarrow \ \text{inc} \ \nu \ \ k)
\end{align*}
\]
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 \( e_1 \).

\[ C[e_1 + e_2]_k \Rightarrow C[e_1](\lambda v \rightarrow k(v + e_2)) \text{ where } v \text{ is fresh.} \]

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

\[ C[e_1 + e_2]_k \Rightarrow C[e_1](\lambda v_1 \rightarrow C[e_2](\lambda v_2 \rightarrow k(v_1 + v_2))) \text{ where } v_1 \text{ and } v_2 \text{ are fresh.} \]

Notice that we need to nest the continuations!
Examples

\[
\begin{align*}
\text{foo } a \ b &= a + b \\
\text{bar } a \ b &= \text{inc } a + b \\
\text{baz } a \ b &= a + \text{inc } b \\
\text{quux } a \ b &= \text{inc } a + \text{inc } b \\
\end{align*}
\]

\[
\begin{align*}
\text{foo } a \ b \ k &= k \ a + b \\
\text{bar } a \ b \ k &= \text{inc } a (\ v \rightarrow \ k \ (v + b)) \\
\text{baz } a \ b \ k &= \text{inc } b (\ v \rightarrow \ k \ (a + v)) \\
\text{quux } a \ b \ k &= \text{inc } a (\ v1 \rightarrow \ \text{inc } b (\ v2 \rightarrow \ k \ (v1 + v2))) \\
\end{align*}
\]
Other Topics

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