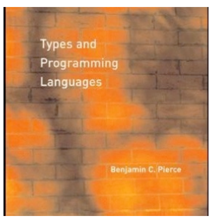
Continuations and Evaluation Contexts

CS 565
Lecture 6



Continuations

Evaluation contexts provide a syntactic formulation to specify evaluation order.

Can we understand evaluation contexts from the perspective of language-level constructs?

- How does one reify the notion of a “hole” in a program?

Continuations and CPS

Starting point:

- How do we represent a program's control-flow?
  - *Loops*
  - *Procedure call and return*
  - *Order of evaluation*
- Our previous semantic characterizations kept these notions implicit.
- Can we make these sequences explicit?
  - *How can we express evaluation contexts within a program?*

Example: Recursion

Consider a factorial function:

```
fun fact(n:int):int = if n = 0
                      then 1
                      else n * fact(n-1)
```

Each call to fact is made with a "promise" that the value returned will be multiplied by the value of n at the time of the call.

Example: Tail Recursion

Now, consider:

```
let fun fact-iter(n:int):int =
   let fun loop(n:int,acc:int):int =
            if n = 0
            then acc
            else loop(n - 1, n * acc)
   in loop(n,1)
   end
```

There is no promise made in the call to loop by fact-iter, or in the inner calls to loop: each call simply is obligated to return its result. Unlike fact, no extra control state (e.g., promise) is required; this information is supplied explicitly in the recursive calls.
What is the implication of these different approaches?
  Recursive vs. iterative control

Tail position

An expression in tail position requires no additional control-information to be preserved.

- Intuitively, no state information needs to be saved.
- Examples:

  The true and false branches of an if-expression.

  A loop iteration.

  A function call that occurs as the last expression of its enclosing definition.
- Tail recursive implementations can execute an arbitrary number of tail-recursive calls without requiring memory proportional to the number of these calls.

Continuation-passing style

Is a technique that can translate any procedure into a tail recursive one.

Example:

```
4 * 3 * 2 * fact(1)
```

- Define the context of fact(1) to be

```
fun Context(v:int):int = 4 * 3 * 2 * v
```

- The context is a function that given the value produced by `fact(1)` returns the result of `fact(4)`

Example revisited

```
fun fact-cps(n:int, k: int -> int): int =
   if n = 0
   then k(1)
   else fact-cps(n-1, fn v => k (n * v))
```

The `k` represents the *function's continuation*: it is a function that given a value returns the "rest of the computation"

By making `k` explicit in the program, we make the control-flow properties of `fact` also explicit, which will enable improved compiler decisions.

Observe that `k(fact(n))=fact-cps(n,k)` for any `k`.

Example revisited

```
fact-cps(4,k) ⟶

 fact-cps(3, fn v => k(4 * v))
 fact-cps(2, fn v => (fn v => k(4 * v))(3 * v))
 fact-cps(2, fn v => (fn v => k(4 * v))(3 * v))
 fact-cps(2, fn v =>  k ( 4 * 3 * v))
 fact-cps(1, fn v => (fn v => k ( 4 * 3 * v)) (2 * v))
 fact-cps(1, fn v => (fn v => k ( 4 * 3 * v))(2 * v))
 fact-cps(1, fn v => k (4 * 3 * 2 * v))
                 …
 fact-cps(0, fn v => k (4 * 3 * 2 * 1 * v))
 (fn v => k (4 * 3 * 2 * 1 * v)) 1
 k 24
```

The initial k supplied to fact-cps represents the “context” in which the call was made.

First cut

Start with a very simple language:

- Variables, functions, applications, and conditionals.

Define a translation function:

```
C : Exp x Cont ⟶ Exp
```

- A continuation will be represented as a function that takes a single argument, and performs “the rest of the computation”
- The translation will ensure that functions never directly return -- they always invoke their continuation when they have a value to provide.
- In an interpreter, the top-level continuation simply prints the value passed as argument to the screen.

The CPS Algorithm

$$C[x]\ k\ =\ k\ x$$

Returning the value of a variable simply passes that value to the current continuation.

$$C[\lambda x.e]\ k\ =\ k\ (\lambda x.\lambda k_2.C[e]\ k_2)$$

A function takes an extra argument which represents the continuation(s) of its call point(s), and its body is evaluated in this context.

$$C[e_1(e_2)]\ k\ =\ C[e_1]\ (\lambda v.C[e_2]\ \lambda v_2.v\ v_2\ k)$$

An application evaluates its first argument in the context of a continuation that evaluates its second argument in the context of a continuation that performs the application and supplies the result to its context.

The Initial Algorithm

$$\begin{aligned}&C[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\ k\ =\\ &\qquad C[e_1]\ (\lambda v.\text{if } v \text{ then } C[e_2]\ k \text{ else } C[e_3]\ k)\end{aligned}$$

Evaluate the test expression in a context that evaluates the true and false branch in the context of the conditional.
Note that `k` is duplicated in both branches. We would like to avoid this.

```
C[if true then x else y] k              =
C[true](λv.if v then C[x]k else C[y]k) =
C[true](λv.if v then k x else k y )    =
(λv.if v then k x else k y )true       →
k x
```

Example

$\texttt{C[(plus1 x)] k} =$

$\texttt{C[plus1]} \; (\lambda \texttt{v.C[x]} \; (\lambda \texttt{v}_2\texttt{.v v}_2 \texttt{ k})) =$

$(\lambda \texttt{v.C[x]} \; (\lambda \texttt{v}_2\texttt{.v v}_2 \texttt{ k})) \; \texttt{plus1} =$

$(\lambda \texttt{v.}(\lambda \texttt{v}_2\texttt{.v v}_2 \texttt{ k}) \; \texttt{x}) \; \texttt{plus1} =$

$(\lambda \texttt{v.}(\lambda \texttt{v}_2\texttt{.v v}_2 \texttt{ k}) \; \texttt{x}) \; \texttt{plus1} \rightarrow$

$(\lambda \texttt{v}_2\texttt{. plus1 v}_2 \texttt{ k}) \; \texttt{x} \rightarrow$

$\texttt{plus1 x k} \rightarrow$

What's plus1 in CPS?

Example

What's plus1 in CPS form?

$\lambda \texttt{x.succ x} =$

$\lambda \texttt{x.}\lambda \texttt{k.C[succ x] k} =$

$\lambda \texttt{x.}\lambda \texttt{k.C[succ]} \; (\lambda \texttt{v.C[x]}(\lambda \texttt{v}_2\texttt{.v v}_2 \texttt{ k})) =$

$\lambda \texttt{x.}\lambda \texttt{k.}(\lambda \texttt{v.C[x]}(\lambda \texttt{v}_2\texttt{.v v}_2 \texttt{ k}))\texttt{succ} =$

$\lambda \texttt{x.}\lambda \texttt{k.}(\lambda \texttt{v.}(\lambda \texttt{v}_2\texttt{.v v}_2 \texttt{ k})\texttt{x})\texttt{succ} \rightarrow$

$\lambda \texttt{x.}\lambda \texttt{k.}(\lambda \texttt{v.}(\texttt{v x k}))\texttt{succ} \rightarrow$

$\lambda \texttt{x.}\lambda \texttt{k.}(\texttt{succ x k})$

Correspondence

What do continuations (and CPS) have to do with evaluation contexts?

- CPS also fixes evaluation order.
- Is a continuation a representation of an evaluation context?

  It represents an expression with a hole (its argument). When supplied a value, the hole is “plugged” and a new expression is produced.

  Not quite: CPS produces continuations that given an intermediate result return the final result of the computation.

  Two incompatible definitions:

  - a context expects an expression, and returns another expression
  - a continuation expects a value and returns a final result.