

Continuations and Continuation-Passing Style

Lecture 4

CS 390

1/16/08

Goal

- We're interested in understanding how to represent the state of a co-routine
 - Insight into what a thread really means
- How fundamental are co-routines (and threads)?
- Key questions:
 - What do we mean by a program's execution state?
 - Can this be captured?
 - Is there a way to represent this state directly within a program?

Continuation-Passing Style

- Starting point:
 - How do we represent a program's control-flow?
 - Loops
 - Procedure call and return
 - Programming languages abstract these notions as primitives:
 - We don't see the "instruction sequence" that represents a procedure call or a loop iteration.
 - Can we make these sequences explicit?

Example

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 (cont)

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:
 - 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:

$4 * 3 * 2 * \text{fact}(1)$

- Define the `context` of `fact(1)` to be

$\text{fn } v \Rightarrow 4 * 3 * 2 * v$

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 $k(\text{fact}(n)) = \text{fact-cps}(n, k)$ 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))      by def. of fact-cps
  fact-cps(2, fn v => k ( 4 * 3 * v))                by beta-conversion
  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.

Our Goal

- Take program and convert it to CPS form.
- Issues:
 - Where do we insert continuations?
 - How do we record the “rest of the computation” that a continuation is to represent?
 - How do we distinguish between continuations that
 - Represent the return point of an arbitrary procedure call (e.g., the outer call to fact-cps).
 - Represent iterative computation (e.g., the inner recursive calls in fact-cps)
- Why?
 - If we can represent a program’s control at any point, then we can use it to suspend and resume the program.
 - This provides the necessary infrastructure to implement co-routines.

First cut

- Start with a very simple language:
 - Variables, functions, applications, and conditionals.
- Define a translation function:
 - $C : \text{Exp} \times \text{Cont} \rightarrow \text{Exp}$
 - A continuation will be represented as a function that takes a single argument, and perform “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.

The Initial Algorithm

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

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

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

A function takes an extra argument which represents the continuation(s) of its call point(s), and its body is evaluated in this context. Lambda (λ) notation is used as a shorthand for function definition.

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

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.

Initial Algorithm (cont)

$C [\text{if } e1 \text{ then } e2 \text{ else } e3] k =$

$C[e1] \lambda v. \text{if } v \text{ then } C[e2]k \text{ else } C[e3]k$

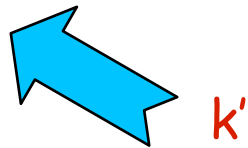
Evaluate the test expression in a context that evaluates the true and false branch in the context of the conditional.

Example

$C [(x1(x2) x3)] k \rightarrow$

$C [x1(x2)] \lambda v1 . C [x3] \lambda v2 . v1(v2,k) \rightarrow$

$C [x1(x2)] \lambda v1 . (\lambda v2 . v1(v2,k)) x3 \rightarrow$



$C [x1] \lambda v3 . C [x2] \lambda v4 . v3(v4,k') \rightarrow$

$(\lambda v3 . (\lambda v4 . v3(v4,k')) x2)$

$x1)$

Example (cont)

$C [x1] \lambda v3 . C [x2] \lambda v4 . v3(v4, k') \quad \rightarrow$

$(\lambda v3 . (\lambda v4 . v3(v4, k') x2) x1) \quad \rightarrow$

$(\lambda v3 . (\lambda v4 . v3(v4, k') x2) x1) \quad \rightarrow$

$x1(x2, k') \quad \rightarrow$

$x1(x2, (\lambda v1 . (\lambda v2 . v1(v2, k) x3)) x3)) \quad \rightarrow$

Continuations

- CPS provides a translation mechanism that generalizes control structures
- Can we reify this notion into a source language?
 - result is a continuation, a reified representation (in the form of an abstraction) of a program control-stack.
 - Define a primitive operation called call/cc:
 - call-with-current-continuation
 - callcc (fn k => e)
 - captures the current continuation, binds to k, and evaluate e
 - throw k x
 - apply continuation k with argument x

Examples

`call/cc (λ k. (k 3) + 2) + 1 → 4`

`val r = ref (λ v. 0)`

`call/cc (λ k. (r := k; (k 3) + 2)) + 1 → 4`

`(!r 4) → 5`

`let f = call/cc (λ k. λ x. k (λ y. x + y))`

`in f 6 →`

`12`

What have we achieved?

- Call/cc gives us a way to capture the remaining part of a computation at any given program point.
- If we capture a continuation and store it, we have a handle on a program state. This is what is necessary to suspend a co-routine.
- If we invoke a captured continuation, we effectively resume a computation (or co-routine) at the point where its continuation was saved.

References

- A good introductory reference on CPS can be found in Friedman, Wand, Haynes, *Essentials of Programming Languages*.
- You can also find many good tutorials on the Web.