

Continuations

CS242

Lecture 6

HW1 Comments

- People were very engaged with HW1
 - Office hours were busy!
- Overall people did well
- Recurring issues
 - Expressions are trees, not strings.
 - Example $S K I I$ cannot be reduced to $S I$
 - Data type encoding

Today's Variant of Lambda Calculus ...

$$e \rightarrow x \mid \lambda x.e \mid e e \mid i \mid e + e$$

Start Simple

- How do we evaluate $e + e'$?
- First evaluate e to a value x
- Second evaluate e' to a value y
- Third compute $x + y$
- Note that this description fixes an order of evaluation
 - Could evaluate e' and then e instead

Explicit Order of Evaluation

We can rewrite the expression to make the order of evaluation explicit:

$$(\lambda x. (x + e') \ e)$$

Going one step further:

$$(\lambda x. ((\lambda y. x + y) \ e')) \ e$$

Explicit Order of Evaluation

We can rewrite $e + e'$ to make the order of evaluation explicit:

$$(\lambda x. (x + e') \ e)$$

Going one step further:

$$(\lambda x. ((\lambda y. x + y) \ e')) \ e$$

And one more step:

$$(\lambda x. ((\lambda y. (\lambda z. z) \ (x + y)) \ e')) \ e$$

A More Readable Version

Recall

$(\lambda x. e) \ e' = \text{let } x = e' \text{ in } e$

Then $(\lambda x. ((\lambda y. (\lambda z. z) (x + y)) e')) \ e$

Can be rewritten as

let $x = e$ in

let $y = e'$ in

let $z = x + y$ in

z

Comments

```
let x = e in  
  let y = e' in  
    let z = x + y in  
      z
```

Can be read as a sequential program

```
x = e  
y = e'  
z = x + y
```


Comments

```
let x = e in
  let y = e' in
    let z = x + y in
      z
```

Note:

- The order of evaluation is explicit
- Every intermediate result has a name

Back to the First Step

Recall the first step of the transformation:

$$(\lambda x. x + e') \ e$$

Which is equivalent to

$$\text{let } x = e \text{ in } e'$$

Continuations

let $x = e$ in e'

We can view this as splitting the program into two sequentially ordered parts:

- The computation of $x = e$
- The *continuation* e' which represents the computation of the rest of the program

What is a Continuation?

Recall $\text{let } x = e \text{ in } e' \Leftrightarrow (\lambda x. e') e$

A continuation is a function that takes a value as an argument and evaluates the “rest of the program”.

Continuation Passing Style

- Rewrite the program using continuations
- Each continuation
 - Performs just one primitive step of the computation
 - And then passes the result to another continuation

Back to the Example

Recall we translated $e + e'$ to

$$(\lambda x. ((\lambda y. (\lambda z. z) (x + y)) e')) e$$
$$k_0 = \lambda w. k_1 e$$
$$k_1 = \lambda x. k_2 e'$$
$$k_2 = \lambda y. k_3 (x + y)$$
$$k_3 = \lambda z. z$$

Back to the Example

Recall we translated $e + e'$ to

$(\lambda x. ((\lambda y. (\lambda z. z) (x + y)) e')) e$

$k_0 = \lambda w. k_1 e$

$k_1 = \lambda x. k_2 e'$

$k_2 = \lambda y. k_3 (x + y)$

$k_3 = \lambda z. z$

k_0 : let $x = e$ in

k_1 : let $y = e'$ in

k_2 : let $z = x + y$ in

k_3 : z

Continuations

- Continuations are like statement labels in C
 - Syntactically, names a point in the program
 - Semantically, names the computation that executes by jumping to that point
- By systematically using continuations, we
 - Make the order of evaluation explicit
 - Give a name to every intermediate value
 - *Name every step (continuation) of the computation*

Continuation Passing Style Transformation

Define $C(e,k)$ to be the translation of e with continuation k into continuation passing style

So semantically, $C(e,k) = k\ e$

i.e., evaluate e and pass the value to k to run the rest of the program.
But of course we want to convert e into CPS style, too ...

CPS Transformation: Constants and Variables

Easy cases first!

$$C(i, k) = k \ i$$

$$C(x, k) = k \ x$$

For an integer or variable there is no further translation to do, just pass the value directly to the continuation.

CPS Transformation: Addition

$$C(e + e', k) = C(e, \lambda v. C(e', \lambda v'. k (v + v')))$$

Note: The variables v and v' must be fresh.

CPS Transformation: Abstraction

$$C(\lambda x.e, k) = ?$$

Here k is the continuation of the function definition.

We also want to translate the body e of the function. What is the continuation of the function body?

Problem: The function is called at a different point than it is defined, so the continuation for the body is different and not an argument to the translation.

CPS Transformation: Abstraction

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

Idea: Simply define the translation of the function to first take a continuation k' and then take the function argument.

The continuation when the function is applied is k' , which we use in the translation of the function body.

Notice how the two continuations k and k' capture the two relevant points in a function's life: When it is defined and when it is applied.

CPS Transformation: Application

$$C(e\ e', k) = C(e, \lambda f. C(e', \lambda v. f\ k\ v))$$

The translation is fully determined by two things:

We evaluate e and then e' . Note the structural similarity to addition, the other construct with two subexpressions.

The expression e evaluates to a CPS-transformed function f , requiring a continuation k and a value v as arguments.

Continuation Passing Style Transformation

$$C(x, k) = k \ x$$

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

$$C(e \ e', k) = C(e, \lambda f. C(e', \lambda v. f \ k \ v))$$

$$C(i, k) = k \ i$$

$$C(e + e', k) = C(e, \lambda v. C(e', \lambda v'. k \ (v + v')))$$

Reminder

When reading lambda expressions, the scope of an abstraction $\lambda x.e$ extends as far to the right as possible

- All the way to the end of the expression
- Or until blocked by a right parenthesis

$\lambda f.\lambda x.\lambda y. f\ y\ x = \lambda f.\lambda x.\lambda y. (f\ y\ x)$

is very different from

$\lambda f.\lambda x.(\lambda y. f\ y)\ x$

An Example

$$C((\lambda x. x + 1) \ 2, k_0) =$$

$$C(\lambda x. x + 1, \lambda f. C(2, \lambda v_0. f \ k_0 \ v_0)) =$$

$$C(\lambda x. x + 1, \lambda f. ((\lambda v_0. f \ k_0 \ v_0) \ 2)) =$$

$$(\lambda f. ((\lambda v_0. f \ k_0 \ v_0) \ 2)) \ \lambda k_1. \lambda x. C(x + 1, k_1) =$$

$$(\lambda f. ((\lambda v_0. f \ k_0 \ v_0) \ 2)) \ \lambda k_1. \lambda x. C(x, \lambda v_1. C(1, \lambda v_2. k_1 \ v_1 + v_2)) =$$

$$(\lambda f. ((\lambda v_0. f \ k_0 \ v_0) \ 2)) \ \lambda k_1. \lambda x. C(x, \lambda v_1. (\lambda v_2. k_1 \ v_1 + v_2) \ 1) =$$

$$(\lambda f. ((\lambda v_0. f \ k_0 \ v_0) \ 2)) \ \lambda k_1. \lambda x. (\lambda v_1. (\lambda v_2. k_1 \ v_1 + v_2) \ 1) \ x$$

Evaluation

$(\lambda f.((\lambda v_0.f\ k_0\ v_0)\ 2))\ \lambda k_1.\lambda x.(\lambda v_1.(\lambda v_2.\ k_1\ v_1+v_2)\ 1)\ x \rightarrow$

$(\lambda v_0.(\lambda k_1.\lambda x.(\lambda v_1.(\lambda v_2.\ k_1\ v_1+v_2)\ 1)\ x)\ k_0\ v_0)\ 2 \rightarrow$

$(\lambda k_1.\lambda x.(\lambda v_1.(\lambda v_2.\ k_1\ v_1+v_2)\ 1)\ x)\ k_0\ 2 \rightarrow$

$(\lambda x.(\lambda v_1.(\lambda v_2.\ k_0\ v_1+v_2)\ 1)\ x)\ 2 \rightarrow$

$(\lambda v_1.(\lambda v_2.\ k_0\ v_1+v_2)\ 1)\ 2$

$(\lambda v_2.\ k_0\ 2+v_2)\ 1$

$k_0\ 2+1$

$k_0\ 3$

Complete Programs

For a full program P , the initial continuation is the identify function I .

So the CPS transformation of P is

$\text{CPS}(P, I)$

Discussion

- The CPS transformation is important in language implementations
 - Very convenient to have a program representation where every intermediate result is named.
- But we can go a step further and make it useful to the programmer
 - By making continuations available as program values

Call/CC

$e \rightarrow x \mid \lambda x.e \mid e e \mid i \mid e + e \mid \text{call/cc } \lambda k.e \mid \text{resume } k e$

Call/cc calls its function argument with the current continuation.
Resume passes the value of its expression argument to its continuation argument.

Call/CC

$$C(x, k) = k \ x$$

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

$$C(e \ e', k) = C(e, \lambda f. C(e', \lambda v. f \ k \ v))$$

$$C(i, k) = k \ i$$

$$C(e + e', k) = C(e, \lambda v. C(e', \lambda v'. k \ (v + v')))$$

$$C(\text{call/cc } \lambda x. e, k) = (\lambda x. C(e, k)) \ k$$

$$C(\text{resume } k \ e, k') = C(e, k)$$

Example

call/cc $\lambda k. 1 + (\text{resume } k \ 0)$

What is the result of this program?

Translation and Evaluation

$$\begin{aligned} &C(\text{call/cc } \lambda k.1 + (\text{resume } k \ 0), I) = \\ &(\lambda k.C(1 + (\text{resume } k \ 0), I)) \ I = \\ &(\lambda k.C(1, \lambda m.C(\text{resume } k \ 0, \lambda n.I \ (m + n)))) \ I = \\ &(\lambda k.C(1, \lambda m.C(0, k))) \ I = \\ &(\lambda k.C(1, \lambda m.k \ 0)) \ I = \\ &(\lambda k.(\lambda m.k \ 0) \ 1) \ I \rightarrow \\ &(\lambda m. \ I \ 0) \ 1 \rightarrow \\ &I \ 0 \rightarrow \\ &0 \end{aligned}$$

Discussion

This program simulates an “abort” or “exit” statement

- Capture the continuation at the start of the program
- Invoking that continuation at any point will terminate the computation

Discussion

- In general continuations can be used to resume execution from an arbitrary point in the program
- Can implement many non-local control operations
 - Exceptions
 - Backtracking
 - Setjmp/longjmp
 - Co-routines
 - ...

Discussion

- A few languages expose call/cc or something similar
 - Scheme, Racket
- But programmers can also code continuation-passing style directly
 - Often used as a software architecture device
 - E.g., event-driven systems
- Pluses and minuses
 - Makes program control into first-class values, which is necessary for programs that need to programmatically manipulate the flow of control
 - Turns programs “inside out”
 - Contagious: Affects the structure of the entire program