Continuations

An overview

Outline

- The Continuation Passing Style
- Use in denotational semantics
- The escape operator (call/cc)
- History
- A sneak preview of the continuation monad

The “direct” style

```
fac n = if n == 0 then 1 else n * fac (n - 1)

Using eager evaluation:

1)  fac 2
2)  if 2 == 0 then 1 else 2 * fac (2 - 1)
3)  if False then 1 else 2 * fac (2 - 1)
4)  2 * fac (2 - 1)
5)  2 * fac 1
6)  2 * (if 1 == 0 then 1 else 1 * fac (1 - 1))
7)  2 * (if False then 1 else 1 * fac (1 - 1))
8)  2 * (1 * fac (1 - 1))
9)  2 * (1 * fac 0)
10) 2 * (1 * (if 0 == 0 then 1 else 0 * fac (0 - 1)))
11) 2 * (1 * (if True then 1 else 0 * fac (0 - 1)))
12) 2 * (1 * 1)
13) 2 * 1
14) 2
```

The continuation passing style

```
fac n k = if n == 0 then k 1 else fac (n - 1) (\r -> k (n * r))

1)  fac 2 id
2)  if 2 == 0 then id 1 else fac (2 - 1) (\r1 -> id (2 * r1))
3)  if False then id 1 else fac (2 - 1) (\r1 -> id (2 * r1))
4)  fac (2 - 1) (\r1 -> id (2 * r1))
5)  fac 1 (\r1 -> id (2 * r1))
6)  if 1 == 0 then ...k... 1 else fac (1 - 1) (\r2 -> (\r1 -> id (2 * r1)) (1 * r2))
7)  if False then ...k... 1 else fac (1 - 1) (\r2 -> (\r1 -> id (2 * r1)) (1 * r2))
8)  fac (1 - 1) (\r2 -> (\r1 -> id (2 * r1)) (1 * r2))
9)  fac 0 (\r2 -> (\r1 -> id (2 * r1)) (1 * r2))
10) if 0 == 0 then (\r2 -> (\r1 -> id (2 * r1)) (1 * r2)) 1 else ...
11) if True then (\r2 -> (\r1 -> id (2 * r1)) (1 * r2)) 1 else ...
12) (\r2 -> (\r1 -> id (2 * r1)) (1 * r2)) 1
13) (\r1 -> id (2 * r1)) (1 * 1)
14) (\r1 -> id (2 * r1)) 1
15) id (2 * 1)
16) id 2
```

The continuation passing style, notes

- We could (maybe should) transform operators (*, ==, -) into CPS too.
- Need to be careful with if-then-else, because it needs to be “lazy”.
- The continuation is a reified context.
- The benefit is that can “do things” with the context, now that it is a value:
  - ignore it and replace with something else,
  - modify it,
  - save a copy of it.

The continuation passing style, more notes

- CPS flattens nested expressions, making evaluation order explicit.
- This flattening has strong connections with the kind of transformation you might do in a compiler for an imperative language, to manage control flow! See: Richard Kelsey “A Correspondence between Continuation Passing Style and Single Static Assignment Form” (1995).
- Functions in CPS style never “return” a value, they just call (jump to) their continuation.
- All calls become tail calls.
- You can do away with the call stack, but beware: the continuation may grow, just like the stack. (how to replace one space leak with another).

Plotkin's CPS transform for call-by-value LC

$$[[x]] = \k.\ k\ x$$

$$[[\x.M]] = \k.\ k\ (\x.\ [[M]])$$

$$[[M\ N]] = \k.\ [[M]]\ (\m.\ [[N]]\ (\n.\ (m\ n)\ k))$$

Plotkin, G. D. (1975). Call-by-name, call-by-value and the lambda calulus. Theoretical Computer Science, 1:125-159.

Plotkin's CPS transform for call-by-name LC

$$[[x]] = x$$

$$[[\x.M]] = \k.\ k\ (\x.\ [[M]])$$

$$[[M\ N]] = \k.\ [[M]]\ (\m.\ (m\ [[N]])\ k)$$

Plotkin, G. D. (1975). Call-by-name, call-by-value and the lambda calulus. Theoretical Computer Science, 1:125-159.

Problems with the simple CPS transforms

- Introduce new “administrative” (or plumbing) redexes, which need to be eliminated in a second pass.
- See: Danvy and Filinski “Representing control: a study of the CPS transform” (1992).
- Call by need is more complicated (have to deal with sharing). See: Okasaki, Lee, Tarditi “Call-by-need and Continuation-passing Style” (1993).
- Introduces lots of closures into the transformed program, which can be costly for compilation. See: Steele Jr “Rabbit: a compiler for Scheme” (1978).

Definitional interpreters

- You can apply the CPS transformation to programs as we have seen.
- Or you can transform your interpreter to use continuation passing.
- See Reynolds, “Definitional Interpreters for Higher-Order Programming Languages” (1972)
- In either case, one useful outcome is that the evaluation order of the *defined* language is independent of the evaluation order of the *defining* language.
- Eg, suppose you implement an interpreter for Scheme in Haskell. If you use CPS, then the laziness of Haskell does not imply laziness in your Scheme (this requires you to choose the CPS for call-by-value of course). This is a problem in “meta circular” interpreters.

Direct denotational semantics

- The meaning of programs is denoted by “values” from a “semantic domain”.
- Contrast with the operational semantics, where the meaning of programs is given by “actions” (transitions in a state machine).
- The “direct” denotational semantics uses a semantic function:

```
eval :: Term -> Environment -> Value
```

- The denotational semantics is (supposed to be) compositional: the meaning of a compound term is constructed from the meanings of its sub-terms.

Continuation semantics

- The direct style makes it awkward to handle “non-compositional” things like errors, and exotic control flow operators (jumps).

- The same problem happens in the “direct style” of programming in a pure functional language like Haskell. Consider the propagation of “Nothing” (aka failure encoded as a value) upwards through some deeply nested computation.

- Did anyone say “monads”? Note: monadic style is not “direct”.

- An alternative is to use the “continuation semantics”:

```
cont = Value -> Value

eval :: Term -> Environment -> Cont -> Value
```

Continuation semantics and exotic control operators

- The classic example is call/cc from Scheme.

- The expression `callcc e` evaluates the expression `e` to obtain a function, and then applies that function to the current continuation (as well as giving the current continuation to the function as its continuation). (Reynolds “Theories of programming languages” (page 255).

- The expression `throw e1 e2` evaluates `e1` to obtain a continuation and then evaluates `e2`, giving it the continuation that is the value of `e1`, rather than the continuation that is given to the throw expression itself. (*ibid* page 256)

    eval(`callcc e`, env, k) = eval(`e`, env, \f. f k k)

    eval(`throw e1 e2`, env, k) = eval(`e1`, env, \c. eval(`e2`, env, c))

An example of the semantics of callcc

What is the meaning of the term:

```
callcc (\c. 2 + throw c (3 * 4))
```

```
  eval( callcc (\c. 2 + throw c (3 * 4)), k)
= eval( \c. 2 + throw c (3 * 4), \f. f k k)
= eval(2 + throw k (3 * 4), k)
= eval(2, \i. eval(throw k (3 * 4), \j. k(i + j)))
= (\i. eval(throw k (3 * 4), \j. k(i + j))) 2
= eval(throw k (3 * 4), \j. k(2 + j))
= eval(k, \c. eval(3 * 4, c))
= eval(3 * 4, k)
...
= k 12
```

I'm cheating a bit, by dropping the environment, and skipping some steps.

An example of the semantics of callcc

What is the meaning of the term:

```
callcc (\c. 2 + throw c (3 * 4))
```

```
  eval( callcc (\c. 2 + throw c (3 * 4)), k)
= eval( \c. 2 + throw c (3 * 4), \f. f k k)
= eval(2 + throw k (3 * 4), k)
= eval(2, \i. eval(throw k (3 * 4), \j. k(i + j)))
= (\i. eval(throw k (3 * 4), \j. k(i + j))) 2
= eval(throw k (3 * 4), \j. k(2 + j))
= eval(k, \c. eval(3 * 4, c))
= eval(3 * 4, k)
...
= k 12
```

Apply the rule for callcc to get to the next line.

An example of the semantics of callcc

What is the meaning of the term:

```
callcc (\c. 2 + throw c (3 * 4))
```

```
  eval( callcc (\c. 2 + throw c (3 * 4)), k)
= eval( \c. 2 + throw c (3 * 4), \f. f k k)
= eval(2 + throw k (3 * 4), k)
= eval(2, \i. eval(throw k (3 * 4), \j. k(i + j)))
= (\i. eval(throw k (3 * 4), \j. k(i + j))) 2
= eval(throw k (3 * 4), \j. k(2 + j))
= eval(k, \c. eval(3 * 4, c))
= eval(3 * 4, k)
...
= k 12
```

Apply the rule for function abstraction (not shown here) to get to the next line.

An example of the semantics of callcc

What is the meaning of the term:

```
callcc (\c. 2 + throw c (3 * 4))
```

```
  eval( callcc (\c. 2 + throw c (3 * 4)), k)
= eval( \c. 2 + throw c (3 * 4), \f. f k k)
= eval(2 + throw k (3 * 4), k)
= eval(2, \i. eval(throw k (3 * 4), \j. k(i + j)))
= (\i. eval(throw k (3 * 4), \j. k(i + j))) 2
= eval(throw k (3 * 4), \j. k(2 + j))
= eval(k, \c. eval(3 * 4, c))
= eval(3 * 4, k)
...
= k 12
```

Apply the rule for binary operator application (not shown here) to get to the next line.

An example of the semantics of callcc

What is the meaning of the term:

```
callcc (\c. 2 + throw c (3 * 4))
```

```
  eval( callcc (\c. 2 + throw c (3 * 4)), k)
= eval( \c. 2 + throw c (3 * 4), \f. f k k)
= eval(2 + throw k (3 * 4), k)
= eval(2, \i. eval(throw k (3 * 4), \j. k(i + j)))
= (\i. eval(throw k (3 * 4), \j. k(i + j))) 2
= eval(throw k (3 * 4), \j. k(2 + j))
= eval(k, \c. eval(3 * 4, c))
= eval(3 * 4, k)
...
= k 12
```

Apply the rule for constants (not shown here) to get to the next line.

An example of the semantics of callcc

```
What is the meaning of the term:

callcc (\c. 2 + throw c (3 * 4))


  eval( callcc (\c. 2 + throw c (3 * 4)), k)
= eval( \c. 2 + throw c (3 * 4), \f. f k k)
= eval(2 + throw k (3 * 4), k)
= eval(2, \i. eval(throw k (3 * 4), \j. k(i + j)))
= (\i. eval(throw k (3 * 4), \j. k(i + j))) 2
= eval(throw k (3 * 4), \j. k(2 + j))
= eval(k, \c. eval(3 * 4, c))
= eval(3 * 4, k)
...
= k 12
```

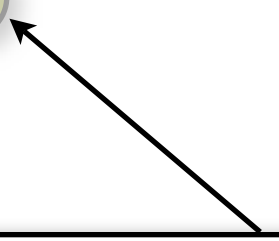
Apply function application to get to the next line. Yes, now I am REALLY cheating.

An example of the semantics of callcc

What is the meaning of the term:

```
callcc (\c. 2 + throw c (3 * 4))
```

```
  eval( callcc (\c. 2 + throw c (3 * 4)), k)
= eval( \c. 2 + throw c (3 * 4), \f. f k k)
= eval(2 + throw k (3 * 4), k)
= eval(2, \i. eval(throw k (3 * 4), \j. k(i + j)))
= (\i. eval(throw k (3 * 4), \j. k(i + j))) 2
= eval(throw k (3 * 4), \j. k(2 + j))
= eval(k, \c. eval(3 * 4, c))
= eval(3 * 4, k)
...
= k 12
```



Apply the rule for throw to get to the next line (notice that it tosses away its continuation).

An example of the semantics of callcc

What is the meaning of the term:

```
callcc (\c. 2 + throw c (3 * 4))
```

```
  eval( callcc (\c. 2 + throw c (3 * 4)), k)
= eval( \c. 2 + throw c (3 * 4), \f. f k k)
= eval(2 + throw k (3 * 4), k)
= eval(2, \i. eval(throw k (3 * 4), \j. k(i + j)))
= (\i. eval(throw k (3 * 4), \j. k(i + j))) 2
= eval(throw k (3 * 4), \j. k(2 + j))
= eval(k, \c. eval(3 * 4, c))
= eval(3 * 4, k)
...
= k 12
```

Apply the rule for variables (not shown) to get to the next line

An example of the semantics of callcc

What is the meaning of the term:

```
callcc (\c. 2 + throw c (3 * 4))
```

```
  eval( callcc (\c. 2 + throw c (3 * 4)), k)
= eval( \c. 2 + throw c (3 * 4), \f. f k k)
= eval(2 + throw k (3 * 4), k)
= eval(2, \i. eval(throw k (3 * 4), \j. k(i + j)))
= (\i. eval(throw k (3 * 4), \j. k(i + j))) 2
= eval(throw k (3 * 4), \j. k(2 + j))
= eval(k, \c. eval(3 * 4, c))
= eval(3 * 4, k)
...
= k 12
```

Apply the rule for binary operators and then constants to get to the bottom.

Delimited continuations

- See: Danvy, Filinski “Representing control: a study of the CPS transformation” (1992).

- Introduce two new operators: `shift` and `reset`.

```
1 + reset(10 + shift c in c (c 100))

=> 1 + (10 + (10 + 100))

=> 121
```

- Allows the definition of many new control operators not available with call/cc (eg nondet choice).

History of the discoveries of continuations

- See: Reynolds “The Discoveries of Continuations” (1993).
- Algol 60 a big motivation for formal definition of programming languages.
- Adriaan van Wijngaarden appears to be the first to describe the CPS transform in 1964, as a compilation technique for imperative languages.
- Lots of influential people heard van Wijngaarden’s talk, but it did not take root.
- However, it did spur Dijkstra to pen “Go to statement considered harmful”.

History of the discoveries of continuations, cont'd

- Wadsworth uses continuations in Strachey's lattice-theoretic semantics to allow imperative features a denotational treatment. (1970)
- Did not publish until 1973
- "Strachey felt that it was often good to live with, and try out, a promising idea for a while before publishing --- not the dominant practice nowadays.... get it reasonably polished before bothering the world with results that may be of transient value."
- Abdali (1973) uses continuations to translate Algol 60 into the untyped lambda calculus.

History of the discoveries of continuations, cont'd

- Reynolds: "the early history of continuations is a sharp reminder that original ideas are rarely born in full generality, and that their communication is not always a simple or straightforward task"

Sneak preview of the continuation monad

- Let’s look at some code in terpie.