15

Continuations

There is a way to think about evaluation that lends itself well to capturing several kinds of non-local control-flow features. The idea is related to the idea of *asynchronous functions* found in some languages such as JavaScript (and especially its NodeJS framework). Intuitively, every function expects an extra argument called a *callback*, and when the function is called, instead of returning a result value to its caller it calls the callback with the result.

In programming language theory, this callback argument expected by every function is called a *continuation*. It is straightforward to turn simple expressions into ones using continuations. Consider `(+ 1 (* 3 4))`. Let K be a continuation that will use the result of this evaluation. Then the translation of the expression with respect to continuation K would be written:

`(* 3 4 (fun (x) (+ 1 x` K`)))`

where `*` and `+` have been modified so that they take that extra continuation argument and call that argument when they have computed their result.

For more complex expressions, constructing a version that uses continuations can be a bit more complex. Consider the recursive `sum` function that computes the sum of all numbers between 0 and an argument n:

```
(fun sum (n) (if (= n 0) 0 (+ n (sum (- n 1)))))
```

Here is one possible transformation into a version which uses continuations — note that function `sum` now takes an extra argument, the continuation:

```
(fun sum (n k)
  (= n 0 (fun (x)
           (if x (k 0)
```

```
(- n 1 (fun (y)
         (sum y (fun (z)
                  (+ n z k)))))))))
```

To call this function, you pass it a continuation that does something with the result.

Three things to note:

1. This is very powerful — at every point during the evaluation, there is a continuation which represents the rest of the computation. It is just a function within the object language, meaning it can be treated like any other value — it can be passed around to other functions, or stored in reference cells. And if we store such a continuation and call it again, we automatically resume evaluation from that point in the evaluation. The full power of this will not become clear until we use it to express non-local control flow.

2. The resulting function `sum` is tail recursive. In fact, every function call in `sum` is in tail position. Technically speaking, we do not need to grow the stack to execute such a function. And indeed, if we write and execute such a function in our interpreter modified to do tail-call optimization, this function executes without a hitch even for large values of `n`.

3. There is an automatic way to transform direct code into code that uses continuations, which means, following (2) above, that there is an automatic way to transform every recursive function into a tail-recursive function.

Code written in such a way that every function expects an extra continuation argument that gets invoked with the result of the function is said to be written in *continuation-passing style* (CPS.

As I mentioned, there is an automated way to translate code into CPS. It is a recursive transformation on the structure of expressions. I write

$$[\![exp]\!]K$$

for the result of transformation of expression *expr* in the context of a con-

tinuation K (also an expression) which expects the result of evaluating *exp*:

$$
\begin{aligned}
[\![v]\!]K &= \texttt{(}K\ v\texttt{)} \\
[\![\texttt{(if } c\ t\ e\texttt{)}]\!]K &= [\![c]\!]\texttt{(fun (x) (if x } [\![t]\!]K\ [\![e]\!]K\texttt{))} \\
[\![id]\!]K &= \texttt{(}K\ id\texttt{)} \\
[\![\texttt{(fun (}x\ \texttt{...)}\ \ e\texttt{)}]\!]K &= \texttt{(}K\ \texttt{(fun (}x\ \texttt{...}\ \ \texttt{k) } [\![e]\!]\texttt{k))} \\
[\![\texttt{(}e\ e_1\texttt{)}]\!]K &= [\![e]\!]\texttt{(fun (f) } [\![e_1]\!]\texttt{(fun (a) (f a } K\texttt{)))} \\
[\![\texttt{(}e\ e_1\ e_2\texttt{)}]\!]K &= [\![e]\!]\texttt{(fun (f)} \\
&\qquad\qquad [\![e_1]\!]\texttt{(fun (a) } [\![e_2]\!]\texttt{(fun (b) (f a b } K\texttt{))))} \\
[\![\texttt{(}e\ e_1\ e_2\ e_3\texttt{)}]\!]K &= \ldots
\end{aligned}
$$

You should be able to generalize the above to functions applied to arbitrary many arguments.

This translates reasonably easily into actual code. Every expression node in the abstract representation has a method `cps()` taking a continuation (again, just another expression in the object language) and returning some new abstract representation expression representing the result of the CPS transformation for that expression node. The one subtlety is that all the parameters to the functions introduced by the translation need to be *fresh*, that is, not appearing anywhere in the code being translated. That is in order to avoid accidental capture of existing identifiers. We do this via the function `gensym` which yields a new name.

```
abstract class Exp {
  ...
  def cps (K : Exp) : Exp
}

class EInteger (val i:Integer) extends Exp {
  ...
  def cps (K : Exp) : Exp =
    new EApply(K, List(this))
}

class EBoolean (val b:Boolean) extends Exp {
  ...
  def cps (K : Exp) : Exp =
    new EApply(K, List(this))
}
```

```
class EIf (val ec : Exp, val et : Exp, val ee : Exp) extends Exp {
  ...
  def cps (K : Exp) : Exp = {
    val n = gensym()
    return ec.cps(new EFunction("", List(n), new EIf(new EId(n), et.cps(K), ee.cps(K))))
  }
}

class EId (val id : String) extends Exp {
  ...
  def cps (K : Exp) : Exp =
    return new EApply(K, List(this))
}

class EApply (val f: Exp, val args: List[Exp]) extends Exp {
  ...
  def cps (K : Exp) : Exp = {
    // create names for expressions
    val names = args.map((e) => gensym())
    val fname = gensym()
    val nargs = names.map((n) => new EId(n))
    var result = f.cps(new EFunction("", List(fname), new EApply(new EId(fname), K::
      nargs)))
    for ((e,n) <- args.zip(names)) {
      result = e.cps(new EFunction("", List(n), result))
    }
    return result
  }
}

class EFunction (val self: String, val params: List[String], val body : Exp) extends Exp {
  ...
  def cps (K : Exp) : Exp = {
    val n = gensyn()
    return new EApply(K, List(new EFunction(self, n::params, body.cps(new EId(n)))))
  }
}

case class ELet (val bindings : List[(String,Exp)], val body : Exp) extends Exp {
  ...
  def cps (K : Exp) : Exp = {
    // create names for expressions
    val names = bindings.map((_) => gensym())
    val nbindings = bindings.zip(names).map({ case ((n,e),nnew) => (n,new EId(nnew))
      })
```

```
    var result : Exp = new ELet(nbindings,body.cps(K))
    for (((n,e),nnew) <- bindings.zip(names)) {
      result = e.cps(new EFunction("", List(nnew), result))
    }
    return result
  }
}
```

How does this help us implement exceptions? The above transforms an expression into one where every function has an extra continuation representing where to send the result of the function. In a world with exceptions, we transform every expression into one in which every function has two extra continuations, a success continuation representing where to send the result of the function if it is a value, and a failure continuation representing where to send the result of the function it if is an exception. Throwing an exception will invoke the failure continuation, while catching an exception will involve setting the failure continuation to be one that can handle the exception. Here is the translation $[\![e]\!]^{K_s}_{K_f}$ now parameterized by a success and failure

continuation K_s and K_f:

$$
\begin{aligned}
[\![v]\!]^{K_s}_{K_f} &= (K_s\ v) \\
[\![(\texttt{if}\ c\ t\ e)]\!]^{K_s}_{K_f} &= [\![c]\!]^{K_1}_{K_f} \\
&\quad \text{where } K_1 = (\texttt{fun (x) (if x } [\![t]\!]^{K_s}_{K_f}\ [\![e]\!]^{K_s}_{K_f}\texttt{))} \\
[\![id]\!]^{K_s}_{K_f} &= (K_s\ id) \\
[\![(\texttt{fun (}x\ \ldots\texttt{)}\ \ e)]\!]^{K_s}_{K_f} &= (K_s\ \texttt{(fun (}x\ \ldots\ \ \texttt{ks kf)}\ [\![e]\!]^{\texttt{ks}}_{\texttt{kf}}\texttt{))} \\
[\![(e\ e_1)]\!]^{K_s}_{K_f} &= [\![e]\!]^{K_1}_{K_f} \\
&\quad \text{where } K_1 = (\texttt{fun (f) } [\![e_1]\!]^{K_2}_{K_f}\texttt{)} \\
&\quad \text{where } K_2 = \texttt{(fun (a) (f a } K_s\ K_f\texttt{))} \\
[\![(e\ e_1\ e_2)]\!]^{K_s}_{K_f} &= [\![e]\!]^{K_1}_{K_f} \\
&\quad \text{where } K_1 = (\texttt{fun (f) } [\![e_1]\!]^{K_2}_{K_f}\texttt{)} \\
&\quad \text{where } K_2 = (\texttt{fun (a) } [\![e_2]\!]^{K_3}_{K_f}\texttt{)} \\
&\quad \text{where } K_3 = \texttt{(fun (b) (f a b } K_s\ K_f\texttt{))} \\
[\![(e\ e_1\ e_2\ e_3)]\!]^{K_s}_{K_f} &= \ldots \\
[\![(\texttt{throw}\ e)]\!]^{K_s}_{K_f} &= [\![e]\!]^{\texttt{(fun (a) (}K_f\ \texttt{a))}}_{K_f} \\
[\![(\texttt{try}\ e_1\ \texttt{catch (}x\texttt{)}\ e_2)]\!] &= [\![e_1]\!]^{K_s}_{K_1} \\
&\quad \text{where } K_1 = (\texttt{fun (}x\texttt{) } [\![e_2]\!]^{K_s}_{K_f}\texttt{)}
\end{aligned}
$$

A key aspect of this approach is that we do not need to change the evaluation mechanism. In effect, the transformation to CPS is a form of abstract represenation transformation.

It is sometimes useful to have a continuation-aware abstract representation, where continuations are given special treatment during evaluation.