14

Exceptions

Exceptions are a form of *non-local control flow*, where the flow of evaluation leaves a given context of evaluation and resumes at some other point in the evaluation. In contrast, all of expressions evaluate locally: applying a function can be thought of as transfering control to the body of the function for evaluation, but the result of that evaluation is returned to the point of application; evaluation of a conditional yields evaluation of one of two expressions defined within the conditional, and so on.

The simplest form of non-local control flow is an exception. Exceptions are a way to abort evaluation of an expression from any point within the expression. That the evaluation of an expression was aborted can be caught and an alternate evaluation performed — this is called catching an exception. From the perspective of the evaluator, throwing or raising an exception is a non-local transfer of control: where exactly evaluation resumes depends on where the exception is caught, which cannot be determined by looking at the expression that throws the exception.

There are two main ways to implement exceptions. The first is the easiest, and corresponds to how one would implement exception-like behavior in a language without exceptions: make every expression evaluate either to a value, or to a special status that signifies evaluation was aborted. This special status is checked at every evaluation, and abortion is propagated until caught. The second way to implement exceptions is with the more complex infrastructure of continuations, which has applications beyond exceptions, and to which we'll return in the next chapter.

We implement a very simple exception system where there is a single kind of exception that carries a value. That value can be arbitrary. In a world where we have structured data such as objects or algebraic datatypes, the

value carried by that exception can carry further information about the kind of exception.

Exceptions are easier to implement in an untyped world, and therefore we start with FUNC as our core language. The first step is to define a new class for the result of evaluation, which now has the possibility of yielding a value or failing. It is a standard union pattern, with case classes so that we can use pattern matching.

```
abstract class Result

case class RValue (val v:Value) extends Result

case class RException (val v: Value) extends Result
```

Expression evaluation now returns a `Result`:

```
abstract class Exp {
  ...
  def eval (env : Env[Value]) : Result
}
```

The implementation of evaluation for the various expressions is as before, except that everytime we evaluated subexpressions, we check the result to see if we need to abort.

```
case class EInteger (val i:Integer) extends Exp {
  ...
  def eval (env:Env[Value]) : Result =
    RValue(new VInteger(i))
}

case class EBoolean (val b:Boolean) extends Exp {
  ...
  def eval (env:Env[Value]) : Result =
    RValue(new VBoolean(b))
}

case class EVector (val es: List[Exp]) extends Exp {
  ...
  def eval (env : Env[Value]) : Result = {
    val vs = es.map((e:Exp) => {
      val res = e.eval(env)
      res match {
```

```
        case RException(_) => return res
        case RValue(v) => v
      }
    })
    return RValue(new VVector(vs))
  }
}

case class EIf (val ec : Exp, val et : Exp, val ee : Exp) extends Exp {
  ...
  def eval (env:Env[Value]) : Result = {
    val cr = ec.eval(env)
    cr match {
      case RException(_) => return cr
      case RValue(cv) => {
        if (!cv.getBool()) {
          return ee.eval(env)
        } else {
          return et.eval(env)
        }
      }
    }
  }
}

case class EId (val id : String) extends Exp {
  ...
  def eval (env : Env[Value]) : Result =
    RValue(env.lookup(id))
}

case class EApply (val f: Exp, val args: List[Exp]) extends Exp {
  ...
  def eval (env : Env[Value]) : Result = {
    val rf = f.eval(env)
    rf match {
      case RException(_) => return rf
      case RValue(vf) => {
        val vargs = args.map((e:Exp) => {
          val res = e.eval(env)
          res match {
            case RException(_) => return res
            case RValue(v) => v
          }
        })
```

```
        if (vf.isPrimOp()) {
          return RValue(vf.applyOper(vargs))
        } else {
          // defined function
          // push the vf closure as the value bound to identifier self
          var new_env = vf.getEnv().push(vf.getSelf(),vf)
          for ((p,v) <- vf.getParams().zip(vargs)) {
            new_env = new_env.push(p,v)
          }
          return vf.getBody.eval(new_env)
        }
      }
    }
  }
}

case class ERecFunction (val self: String, val params: List[String], val body : Exp) extends
      Exp {
  ...
  def eval (env : Env[Value]) : Result =
    RValue(new VRecClosure(self,params,body,env))
}


case class ELet (val bindings : List[(String,Exp)], val ebody : Exp) extends Exp {
  ...
  def eval (env : Env[Value]) : Result = {
    var new_env = env
    for ((n,e) <- bindings) {
      val res = e.eval(env)
      res match {
        case RException(_) => return res
        case RValue(v) => new_env = new_env.push(n,v)
      }
    }
    return ebody.eval(new_env)
  }
}
```

All that remains now is to provide the ability to throw an exception, and catch an exception. To following abstract representation is used to throw an exception, passing in a value:

```
case class EThrow (val e : Exp) extends Exp {
  ...
```

```
  def eval (env : Env[Value]) : Result = {
    val res = e.eval(env)
    res match {
      case RException(_) => return res
      case RValue(v) => return RException(v)
    }
  }
}
```

The natural S-expression surface syntax is `(throw` e`)`.

Catching an expression requires the following: a core expression to evaluate, and an alternate expression to evaluate if the core expression throws an exception. We also need a name that we can use in the alternate expression to refer to the value of the exception.

```
case class ETry (val body : Exp, val param: String, val ctch : Exp) extends Exp {
  ...
  def eval (env:Env[Value]) : Result = {
    val rb = body.eval(env)
    rb match {
      case RException(v) =>
        return ctch.eval(env.push(param,v))
      case RValue(_) =>
        return rb
    }
  }
}
```

One S-expression surface syntax for this is `(try` e `catch (`x`)` e`)`.

Of course, we need to modify every client (such as the shell) to handle the new return type of evaluation.

One obvious modification to the evaluation mechanism of our language is to turn every runtime error (such as cause by supplying the wrong number of arguments to a function) into an exception. This requires changing the result type operations from `Value` to `Result`, and returning an `RException` in case of error. In order for this to be truly useful, having a class of values representing exception types would be most useful here, so that one could distinguish, upon catching an exception, the kind of error that happened and handle it accordingly.

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 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 a function `Names.next()` which yields a new name.

```
object Names {
  var counter = 0
  def next () : String = {
    val c = counter
    counter = counter + 1
    return "   x"+counter
  }
}

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

case class EInteger (val i:Integer) extends Exp {
  ...
  def cps (K : Exp) : Exp =
```

```
    new EApply(K, List(this))
}

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

case class EVector (val es: List[Exp]) extends Exp {
  ...
  def cps (K : Exp) : Exp = {
    // create names for expressions
    val names = es.map((e) => Names.next())
    var result : Exp = new EApply(K, List(EVector(names.map((n) => new EId(n)))))
    for ((e,n) <- es.zip(names)) {
      result = e.cps(new ERecFunction("", List(n), result))
    }
    return result
  }
}

case class EIf (val ec : Exp, val et : Exp, val ee : Exp) extends Exp {
  ...
  def cps (K : Exp) : Exp = {
    val n = Names.next()
    return ec.cps(new ERecFunction("", List(n), new EIf(new EId(n), et.cps(K), ee.cps(K))
      ))
  }
}

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

case 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) => Names.next())
    val fname = Names.next()
    val nargs = names.map((n) => new EId(n))
    var result = f.cps(new ERecFunction("", List(fname), new EApply(new EId(fname), K::
```

```
      nargs)))
    for ((e,n) <- args.zip(names)) {
      result = e.cps(new ERecFunction("", List(n), result))
    }
    return result
  }
}

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

case class ELet (val bindings : List[(String,Exp)], val ebody : Exp) extends Exp {
  ...
  def cps (K : Exp) : Exp = {
    // create names for expressions
    val names = bindings.map((_) => Names.next())
    val nbindings = bindings.zip(names).map({ case ((n,e),nnew) => (n,new EId(nnew))
      })
    var result : Exp = new ELet(nbindings,ebody.cps(K))
    for (((n,e),nnew) <- bindings.zip(names)) {
      result = e.cps(new ERecFunction("", 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 $[\![e]\!]_{K_f}^{K_s}$ now parameterized by a success and failure

continuation K_s and K_f:

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

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