Control Structures

In this section we will discuss a number of advanced features that have to do with controlling the flow of execution. We will start by discussing these features, then talk about possible implementations.

What all features have in common is that in one way or another they disrupt the normal evaluation order.

Exception handling

Exceptions are perhaps the most familiar of the various control operations. We already saw it in the past, let us revisit the key idea. The main construct looks something like this:

```
try e1
catch ...
```

The keyword try fixes the current point in time in the execution of the program, as a point we can return to if things go wrong. We then evaluate `e1`. If `e1` evaluates normally, then we continue normally to what comes after the try. but if it does not, then an exception may be "thrown" which takes us back to the try block and onto the catch part, where typically the exception is compared against specific exception types or classes, and a specific expression is evaluated in its turn.

The important part of the puzzle is that when the exception was thrown, the current "evaluation context" is completely thrown away, and evaluation returns to the context in place when try started out. This is the characteristic behavior of these mechanisms, and we will need to find some way to emulate it.

In particular note that when the exception is thrown, we could be lexically very very far from the try code, deep in some library in an entirely different file, in the middle of doing an addition there. Regardless, evaluation immediately jumps to the try location, cancelling all that work.

Generators

Generators are a feature implemented in numerous languages, Python being amongst them. A generator is a function that is able to "yield" control at specific code points, returning a value in the process. The function's caller can return to the function and continue the computation exactly where it left off. In other words the function is *not* called again. Its computation is continued from where it left off.

We can use this to create "streams" of data, via a generator function that keeps yielding new values. For instance here is a Python generator that would keep producing successive Fibonacci numbers, starting with `a` and `b` as the first two. Recall that Fibonacci numbers are generated by each time adding the two previous numbers.

```
def fib(a, b):
  yield a
  yield b
  while True:
    a, b = b, a + b
    yield b
```

The presence of the keyword yield makes this function a generator. In Python, when the function is called a "generator object" is created. It reacts to a next method, and acts as an iterator: Each time next is called the function execution returns from where it last left off, until the next yield. Then it stops and the valued of that yield is returned.

So in the example above a basic execution would be:

```
f = fib(1, 1)
f.next()          # 1 from "yield a"
f.next()          # 1 from "yield b"
f.next()          # 2 from the last line's "yield b"
f.next()          # 3 from the last line's "yield b"
f.next()          # 5 from the last line's "yield b"
# Could also do `next(f)`
```

- So calling the function creates the object f, and when we call f.next() the function's body starts executing.
- When the yield a is encountered the function returns that value a, and in essence remembers where it was.
- The next time f.next() is called the execution will return to that yield a line, and execute the next line, namely yield b. This immediately returns the value 1.
- The next time execution will enter the while loop, update the values of a and b and yields the new b value.
- The next time execution will continue after that last yield b, going back to the start of the while loop. It will again pause/return at the yield b line, with the new value of b. This can go on forever.

An important note is, that if the function were to actually terminate, it would NOT return. It would instead result in an exception being thrown.

Though the idea of a generator is awesome, it is unfortunate that in Python the syntax gets conflated with that of functions. Because making a function into a generator completely changes the function's behavior. Having a different notation for generators would have been preferable.

A nice feature of Python generators is that they are Iterable. For instance they can be used in for loops:

```
for n in fib(1, 1):
  print n
  if n > 100:
    break
```

This will print the Fibonacci numbers up to 144, then break.

Many languages have similar implementations of generators. The Icon programming language in particular is structured in a way so that every expression in the language is actually a generator. Racket also contains generators, as well as some more elaborate features we will discuss in the rest of this section.

Coroutines

Coroutines are a little bit like generators, but they are used to refer to multiple functions each yielding to the others. A similar design is that of threads, where an external scheduler controls when the functions start and stop. Coroutines are like "cooperative threads". They each yield when they are ready (so for example a coroutine can run forever and block all the others from running).

So instead of having one main function calling its helpers, the helpers cooperate with each other and there is no "main function". Only the coroutines, passing information to each other.

We can use Python generators to implement coroutines. The main ingredient is that we should be able to pass information to the generator. The yield expression in Python actually serves two purposes: It yields control to the caller, returning a value if it so chooses, but also receives a value when function execution resumes. We will see how to use that in this section. You can also learn a lot more about using coroutines in Python from this tutorial[1].

Coroutines are perfect for creating pipes or thinking of finite state machines, where your system can be in any number of states, and a coroutine might represent one of these states. An interpreter might be set up using coroutines as follows:

```
Lexer (reads a line/token)  ---send--->
      Parser (parses tokens produces expression) ---send--->
          Interpreter (evaluates and prints result, then yields)
```

So there is a first "coroutine", the lexer, which is really more of a normal function that keeps sending information to the next one. We often call that a "producer". There is also a "consumer" Interpreter, that just reads a result and then yields till there's more results. There is also "filter" in the middle, the Parser, which yields to the Lexer to get tokens and once it can form an expression sends it to the interpreter.

We will not discuss specific code here, as we'll see some more general techniques involving continuations.

Continuations and call/cc

Continuations are a generic mechanism that allows us to implement all of the above features, and more. It has been around in Racket since the early days of Lisp, and has been used behind the scenes in programming language implementations. Continuations have been used as an implementation technique in many programming

[1] http://www.dabeaz.com/coroutines

languages, but Racket actually exposes them to the programmer as "first-class values".

A continuation is essentially a *suspended computation*. It holds the exact state of the computation at a specific time, and it allows to return to that exact moment in time. In practice it is a function that takes in values and returns values.

The Racket method call/cc, a shortcut for call-with-current-continuation allows the call of a function that takes as input the current continuation. That function can then choose to go back to that current continuation and continue the program execution there. That all sounds confusing, so let us look at an example:

```
(define (f k)
  (+ 2
     (begin (k 3)
            2)))
(define (g k)
  (+ 2 2))

(call/cc f)
(call/cc g)
```

Let's see what happens here. f is a function that takes as input a continuation, k. The continuation k is itself a function that takes in a value and holds the state of the program at the location where call/cc occurs. The function f has two choices:

- It can simply return a value and ignore the continuation k. Then the place of the call/cc call is taken by that value. That's what happens with the function g.
- The other alternative is to call the continuation k, giving it a value as input. Then whatever has been going on in f at to that point is completely ignored, and the value you provided k takes the place of the call/cc call. This is what happens with f. Note that even though f was in the middle of adding something to "2" when (k 3) is called, all that is completely discarded. In essence, we *escape* using k.

We leave it as an exercise to implement the various control structures we described via call/cc. You can hopefully imagine how that might be possible: call/cc gives us a way to turn back to a particular point in time for our program. That's the essence behind most other control structures.

Here is a good article[2] to learn more about using continuations. And you can read a lot more about continuations and various control structures on this site[3], along with the warning about the uses of call/cc.

Implementing Continuations

Implementing continuations is an interesting and non-trivial exercise. We will only describe some key ideas here. But the essence of it is that we need to rethink our whole interpreter structure. For example consider a simple program:

[2] http://matt.might.net/articles/programming-with-continuations--exceptions-backtracking-search-threads-ge
[3] http://okmij.org/ftp/continuations/index.html

```
(+ 2 3)
```

Yes, it's a very simple program. Recall what our typical interpreter would do:

- See that it must perform addition.
- Recursively interpret the first operand (2) to get a value.
- Recursively interpret the second operand (3) to get a value.
- Add the two and return the result.

A continuation-based interpreter needs to allow for the possibility that at any given time it might be interrupted. Every single step along the way must be expressable as a continuation. So for instance the moment when we are about to compute the first operand must be captured (because that first operand might have been a call/cc for example) as a continuation that basically expresses the thought: "when you return for me the value of the first operand, I will then ask for the value of the second operand and then add the two up". Similarly the moment when we have found the result of the first operand and are about to compute the second operand must be captured. In total there are actually 3 continuations related to this expression:

- The continuation at the very beginning, which says "when you tell me the result of the whole addition operation I can continue with the rest of the program".
- The continuation as we're about to compute the first operand (2), which says "you tell me the result of this first operand, and I can continue with the addition operation".
- The continuation as we're about to compute the second operand (3), which says "I already know the value of the first operand, and when you tell me the result of the second operand I know how to continue".

In order to implement any functionality like this, our interpreter needs to change:

- The interpreter will now need to receive an extra argument, namely the continuation k.
- Instead of returning a value, the interpreter calls the continuation giving it as input this value.
- These interpreter recursive calls are all tail calls.

We may call this new "interpreter" interp/k. Our normal interpreter can then be recoved by calling interp/k providing to it as an initial continuation a trivial continuation that simply returns its value:

```
(define (interp env e)
  (define (k v) v)
  (interp/k env e k))
```

Or we can imagine a "prompt" which takes some user input, then interprets it with a continuation set up to print the result and ask for more input, setting itself up as the continuation.

```
;; This is pseudocode
(define (prompt v)
  (print v)
  (interp/k '() (user-input) prompt))
```

There are many variations on this theme.

Let us see now how we might implement interp/k in our basic language:

```
(struct varC (s) #:transparent)
(struct numC (n) #:transparent)
(struct boolC (n) #:transparent)
(struct arithC (op e1 e2) #:transparent)
(struct ifC (tst thn els) #:transparent)
(struct pairC (e1 e2) #:transparent)
(struct carC (e) #:transparent)
(struct cdrC (e) #:transparent)
(struct unitC () #:transparent)
(struct letC (s e1 e2) #:transparent)
(struct funC (arg body) #:transparent)
(struct callC (e1 e2) #:transparent)
(struct callccC (e) #:transparent)

(struct numV (n) #:transparent)
(struct boolV (b) #:transparent)
(struct unitV () #:transparent)
(struct pairV (v1 v2) #:transparent)
(struct closV (env f))
(struct contV (k))
```

Note that we had to add a "continuation" value. We would like to implement call/cc, and in order to do that we need our functions to be able to take continuations as arguments. so we need *continuations as first-class values*.

Things start off pretty easily. The cases that are already values are simple, they just provide that value to the continuation:

```
(define (interp/k env e k)
  (cond
    [(varC? e) (k (lookup (varC-s e) env))]
    [(numC? e) (k (numV (numC-n e)))]
    [(boolC? e) (k (boolV (boolC-b e)))]
    [(unitC? e) (k (unitV))]
    [(funC? e) (k (closV env e))]
    ... ))
```

We now need to describe the remaining constructs. We will start with carC. Recall that what carC needs to do is evaluate its argument down to (hopefully) a pair, then extract that pair. But the computation of its argument may take many steps. What carC needs to do in fact is call the interpreter on the component, passing to it a correct continuation. Basically what we do is: "We tail-recursively call the interp/k on the expression, with a continuation that given a value checks that it is a pair, and if it is then it calls the *current* continuation on the first component of that pair". Phew. Let's see it in code:

```
    [(carC? e)
```

```
 (interp/k env (carC-e e)
           (lambda (v)
             (if (pairV? v)
                 (k (pairV-v1 v))
                 (error "car on non-pair")))))]
```

The cdrC is essentially identical.

Now let's tackle a pairC, and many of the constructs that follow are similar.

- We have to interpret a pairC, on a current continuation k.
- We need to recursively interpret the first component, giving it a certain continuation k1.
- That continuation k1 takes as input the value v1 that is supposed to go in the first component, and recursively calls the interpreter on the second component, giving it another continuation k2.
- That continuation k2 takes the value v2, and it already knows the value v1 from its closure, and creates the pair (pairV v1 v2). It then feeds that into the original continuation k.

It may seem surprising that k appears at the end there, but it makes perfect sense! You were trying to evaluate a pair, and that continuation was the one that was supposed to know what comes next. So it makes sense that it will be the last one to act, once the pair is constructed.

Here's the code:

```
[(pairC? e)
 (interp/k env (pairC-e1 e)
           (lambda (v1)
             (interp/k env (pairC-e2 e)
                       (lambda (v2)
                         (k (pairV v1 v2))))))]
```

The interpretation of arithC is similar and doesn't add any new ideas, but reinforces the theme of pairC:

```
[(arithC? e)
 (interp/k env (arithC-e1 e)
           (lambda (v1)
             (interp/k env (arithC-e2 e)
                       (lambda (v2)
                         (if (and (numV? v1)
                                  (numV? v2))
                             (k (numV ((arithC-op e) (numV-n v1) (numV-n v2))))
                             (error "arithmetic on non-numbers")))))))]
```

The conditional is similarly doing something predictable, as does letC:

```
[(ifC? e)
 (interp/k env (ifC-tst e)
           (lambda (v)
             (if (boolV? v)
                 (let ([rest ((if (boolV-b v) ifC-thn ifC-els) e)])
```

```
                  (interp/k env rest k))
                (error "conditional on non-boolean")))]
   [(letC? e)
    (interp/k env (letC-e1 e)
              (lambda (v1)
                (interp/k (bind (letC-s e) v1 env)
                          (letC-e2 e)
                          k)))]
```

Function calls will be discussed after we talk about call/cc.

So let us talk through what call/cc should do:

- It takes one argument, and that argument must evaluate to a function.
- It will then call that argument, passing the current continuation as the argument.
- That function can call that continuation to escape back to the call/cc context.
- So first of all we need to have continuations as first-class values, so we can add them to the environment for the function call.
- Our callccC will essentially have to do some of the work of callC when it calls its function argument. Not particularly elegant, and it's worth thinking if there's a way around it.
- Our callC needs to also allow for the possibility that the value at the function position is a continuation, as opposed to a closure. It needs to handle both cases.

Here's the code:

```
   [(callccC? e)
    (interp/k env (callccC-e e)
              (lambda (vf)
                (if (closV? vf)
                    (interp/k (bind (funC-arg (closV-f vf))
                                    (contV k)
                                    (closV-env vf))
                              (funC-body (closV-f vf))
                              k)
                    (error "callcc on non-function")))]
```

Now let's look at callC. The idea is simple:

- We interpret the expression at function position in a continuation k1.
- That continuation takes the value vf and interprets the expression at argument position in a continuation k2.
- That continuation takes the value varg, and it already has the value vf in its closure. It then checks to make sure that vf is a closure or a continuation.
- If vf is a function, then it evaluates the body of the function in that closure in the appropriate environment and the *original* continuation k.
- If vf is a continuation, then it simply calls that continuation, providing varg as the value. The continuation k is more or less ignored in this instance. Which is at it should be.

```
[(callC? e)
 (interp/k env (callC-e1 e)
           (lambda (vf)
             (interp/k env (callC-e2 e)
                       (lambda (varg)
                         (cond [(closV? vf)
                                (interp/k (bind (funC-arg (closV-f vf))
                                                varg
                                                (closV-env vf))
                                          (funC-body (closV-f vf))
                                          k)]
                               [(contV? vf)
                                ((contV-k vf) varg)]
                               [else (error "calling non-function")])))))]
```

There is a different approach to the whole topic, involving a macro transformation which turns any program into this "continuation passing style". If you are interested, can read all about it in this page[4].

While we could discuss the implementation of generators, coroutines and exceptions using this continuation-based interpreter, these features are somewhat more advanced.

[4] http://cs.brown.edu/courses/cs173/2012/book/Control_Operations.html