Looping and Recursion

April 9, 2020

Riccardo Pucella

Looping

- Looping is core to many tasks:
  - repeating tasks until a condition is met
  - looping over a structure to perform transformations
  - looping over a structure to accumulate information
  - need the ability to nest loops and be able to stop any of them

- Our language FUNC doesn't do too well with respect to looping
  - We talked a _bit_ about EWhile when doing state, but we didn't give it its due

Loop expression

One way of looping is to provide a suitably generic expression to perform loops:

```
(loop name ((v1 init1)
            (v2 init2)
            ...)
    body)
```

In *body*, you can "jump back" to the next loop iteration with (*name e1 e1…*)

- *e1*, *e2*, … new values for loop variables

Example — summing squares

```
(let ((stop 100))
  (loop sum-squares ((i 0)
                     (sum 0))
    (if (= i stop)
      sum
      (do (print i sum)
          (sum-squares (+ i 1)
                       (+ sum (* i i))))))))
```

Example — summing vector

```
(let ((vec [10 20 30 40 50]))
  (loop sum-vec ((curr vec)
                 (sum 0))
    (if (empty? curr)
      sum
      (do (print curr sum)
          (sum-vec (rest curr)
                   (+ sum (first curr)))))))
```

Example — make vector

```
(let ((start 10)
      (end 50))
  (loop make ((curr end)
              (result empty))
    (if (= curr start)
      (cons start result)
      (make (+ curr -1)
            (cons curr result)))))
```

Example — vector reversal

```
(let ((vec [1 2 3 4 5 6]))
  (loop rev ((curr vec)
             (result empty))
    (if (empty? curr)
      result
      (rev (rest curr)
           (cons (first curr) result)))))
```

Example — nested loops

```
(let ((stop 10))
  (loop L1 ((n 0)
            (result empty))
    (if (= n stop)
      result
      (let ((r (loop L2 ((m 0)
                         (subresult empty))
                 (if (= m n)
                   subresult
                   (do (print n m)
                       (L2 (+ m 1)
                           (cons n subresult)))))))
        (L1 (+ n 1)
            (cons r result))))))
```

Implementation

Add a new abstract representation **ELoop**

Evaluation for ELoop:

- repeatedly loops over the body
- maintain the current loop variable values

Challenge:

- (loop-name …) jumps back to loop start
- (loop-name …) is syntax for application
- you can be deep inside another expression!

VLoop

New kind of value associated with a loop name

- when you apply it, it raises an exception
- "please jump back to the beginning of this loop"

```
class NextIteration(val name: String,
                    val values: List[Value])
                                 extends Exception(name)


class VLoop (val name: String) extends Value {
  ...
  override def apply (args: List[Value]): Value = {
    throw new NextIteration(name, args)
  }
}
```

ELoop

```
class ELoop (val name: String, val init: List[(String, Exp)],
                                             val body: Exp) extends Exp {
  def eval (env: Env): Value = {
    var newEnv = env
    val vars = init.map((p) => p._1)
    var values = init.map((p) => p._2.eval(env))
    while (true) {
      newEnv = env.push(name, new VLoop(name))
      for ((n, v) <- vars.zip(values))
        newEnv = newEnv.push(n, v)
      try {
        return body.eval(newEnv)
      } catch {
        case e: NextIteration =>
                  if (e.name == name) values = e.values else throw e
      }
    }
    return new VBoolean(false) // never gets here
  }
```

Great

We have a reasonably natural loop

If you pay attention, though, you'll notice that

```
(loop L ((i 0) (sum 0))
  (if (= i 10000) sum (L (+ i 1) (+ sum i)))))
```

is basically the same as:

```
((fun L (i sum)
  (if (= i 10000) sum (L (+ i 1) (+ sum i))))
  0 0)
```

BUT if we run the later, we have problems

Detour: function call stack

How function calls are implemented in most languages (including Scala):

- Remember what you want to do after the function call (via a stack)
- Evaluate the function body
- When done, return to what you were doing (popping the stack)

The stack remembers all that's needed to resume evaluation after you return from a function call

Recursion is the same - nothing special

Substitution model

```
len = (fun (v) (if (empty? v) 0 (+ 1 (len (rest v)))))

(len [1 2 3])
= ((fun (v) (if (empty? v) 0) (+ 1 (len (rest v))))) [1 2 3])
= (if (empty? [1 2 3]) 0 (+ 1 (len [2 3])))
= (+ 1 (len [2 3]))
= (+ 1 (if (empty? [2 3]) 0 (+ 1 (len [3]))))
= (+ 1 (+ 1 (len [3])))
= (+ 1 (+ 1 (if (empty? [3]) 0 (+ 1 (len [])))))
= (+ 1 (+ 1 (+ 1 (len []))))
= (+ 1 (+ 1 (+ 1 (if (empty? []) 0 (+ 1 (len (rest []))))))))
= (+ 1 (+ 1 (+ 1 0)))
= (+ 1 (+ 1 1))
=  (+ 1 2)
= 3
```

Tail calls

There's something interesting that happens when the **last** thing a function A does is call another function B

- called a **tail call**

Before calling B, you remember what to do next when you return from B

- what you need to remember is to return
- do we even need to bother remembering?
- we could just have A "continue into" B: when B returns it actually returns from A as well

Tail calls in the substitution model

```
len = (fun (v res)
        (if (empty? v) res (len (rest v) (+ 1 res))))


(len [1 2 3] 0)
= ((fun (v res) (if (empty? v) res (len (rest v)
     (+ 1 res)))) [1 2 3] 0)
= (if (empty? [1 2 3]) 0 (len [2 3] (+ 1 0)))
= (len [2 3] 1)
= (if (empty? [2 3]) 1 (len [3] (+ 1 1)))
= (len [3] 2)
= (if (empty? [3]) 2 (len [] (+ 1 2)))
= (len [] 3)
= (if (empty? []) 3 (len (rest []) (+ 1 3)))
= 3
```

So we can more clever with tail calls

Scala is **not** clever, so we can't rely on Scala

Let's change how evaluation works.

Whenever we are evaluating an expression:

- if the last thing we do is evaluate another expression, don't call the evaluation method
- but instead "continue" into the new expression without calling the method

Sounds impossible. Oh ye of little faith.

Evaluation with tail call optimization

Evaluation of an expression now either returns a value **or** returns the expression to continue evaluation as.

- call this **partial** evaluation

Full evaluation of an expression:

- repeatedly partially evaluate the expression until you get a value

Exp

```
class EvalResult
class Done (val v: Value)
class Continue (val exp: Exp, val env: Env)

class Exp  {
  def evalPartial (env: Env): EvalResult
  def eval (env: Env): Value = {
    var currExp = this
    var currEnv = env
    while (true) {
      val r = currExp.evalPartial(currEnv)
      if (r.isDone())
        return r.getResult()
      currExp = r.getExp()
      currEnv = r.getEnv()
    }
    return new VBoolean(false)  // to satisfy the type checker
  }
}
```

EInteger, EBoolean

```
class EInteger (val i:Int) extends Exp {
  ...
  def evalPartial (env: Env): EvalResult =
    new Done(new VInteger(i))
}

class EBoolean (val b:Boolean) extends Exp {
  ...
  def evalPartial (env: Env): EvalResult =
    new Done(new VBoolean(b))
}
```

EId, EFunction

```
class EId (val id: String) extends Exp {
  ...
  def evalPartial (env: Env): EvalResult =
    new Done(env.lookup(id))
}

class EFunction (val recName: String,
                 val params: List[String], val body: Exp)
                                              extends Exp {
  ...
  def evalPartial (env: Env): EvalResult =
    new Done(new VRecClosure(recName, params, body, env))
}
```

EIf

```
class EIf (val ec: Exp, val et: Exp, val ee: Exp)
                                             extends Exp {
  ...
  def evalPartial (env: Env): EvalResult = {
    val ev = ec.eval(env)
    if (ev.isBoolean()) {
      if (!ev.getBool())
        return new Continue(ee, env)
      else
        return new Continue(et, env)
    } else
      runtimeError("condition not a Boolean")
  }
}
```

EApply

```
class EApply (val fn: Exp, val args: List[Exp])
                                             extends Exp {
  ...
  def evalPartial (env: Env): EvalResult = {
    val vfn = fn.eval(env)
    val vargs = args.map((e:Exp) => e.eval(env))
    return vfn.apply(vargs)
  }
}
```

VRecClosure

```
class VRecClosure (val self: String,
                   val params: List[String],
                   val body:Exp, val env:Env)
                                            extends Value {
  ...
  override def apply (args: List[Value]): EvalResult = {
     if (params.length != args.length)
        runtimeError("wrong number of arguments")
     var new_env = env
     for ((p,v) <- params.zip(args))
        new_env = new_env.push(p,v)
     new_env = new_env.push(self,this)
     return new Continue(body, new_env)
  }
}
```

VPrimitive

```
class VPrimitive (val oper: (List[Value]) => Value)
                                          extends Value {
  ...
  override def apply (args: List[Value]): EvalResult =
     new Done(oper(args))
}
```

That's it!

We can now write loops using tail-recursive functions.

We can still use the (loop …) surface syntax

We can use a parser transformation to turn it into the equivalent function definition and application