PL: Lecture #20

Tuesday, November 12th

Implementing Call by Need

As we have seen, there are a number of advantages for lazy evaluation, but its main disadvantage is the fact that it is extremely inefficient, to the point of rendering lots of programs impractical, for example, in:

```
{bind {{x {+ 4 5}}}
  {bind {{y {+ x x}}}
    y}}
```

we end up adding 4 and 5 twice. In other words, we don't suffer from textual redundancy (each expression is written once), but we don't avoid dynamic redundancy. We can get it back by simply caching evaluation results, using a box that will be used to remember the results. The box will initially hold `#f`, and it will change to hold the VAL that results from evaluation:

```
(define-type VAL
  [RktV  Any]
  [FunV  (Listof Symbol) SLOTH ENV]
  [ExprV SLOTH ENV (Boxof (U #f VAL))] ;*** new: mutable cache field
  [PrimV ((Listof VAL) -> VAL)])
```

We need a utility function to create an evaluation promise, because when an `ExprV` is created, its initial cache box needs to be initialized.

```
(: eval-promise : SLOTH ENV -> VAL)
;; used instead of `eval' to create an evaluation promise
(define (eval-promise expr env)
  (ExprV expr env (box #f)))
```

(And note that Typed Racket needs to figure out that the `#f` in this definition has a type of `(U #f VAL)` and not just `#f`.)

This `eval-promise` is used instead of `ExprV` in eval. Finally, whenever we force such an `ExprV` promise, we need to check if it was already evaluated, otherwise force it and cache the result. This is simple to do since there is a single field that is used both as a flag and a cached value:

```
(: strict : VAL -> VAL)
;; forces a (possibly nested) ExprV promise, returns a VAL that is
;; not an ExprV
(define (strict val)
  (cases val
    [(ExprV expr env cache)
     (or (unbox cache)
         (let ([val* (strict (eval expr env))])
           (set-box! cache val*)
           val*))]
    [else val]))
```

But note that this makes using side-effects in our interpreter even more confusing. (It was true with call-by-name too.)

The resulting code follows.

```
;; A call-by-need version of the SLOTH interpreter

#lang pl

;;; ---------------------------------------------------------------
;;; Syntax

#| The BNF:
   <SLOTH> ::= <num>
             | <id>
             | { bind {{ <id> <SLOTH> } ... } <SLOTH> }
             | { fun { <id> ... } <SLOTH> }
             | { if <SLOTH> <SLOTH> <SLOTH> }
             | { <SLOTH> <SLOTH> ... }
|#

;; A matching abstract syntax tree datatype:
(define-type SLOTH
  [Num  Number]
  [Id   Symbol]
  [Bind (Listof Symbol) (Listof SLOTH) SLOTH]
  [Fun  (Listof Symbol) SLOTH]
  [Call SLOTH (Listof SLOTH)]
  [If   SLOTH SLOTH SLOTH])

(: unique-list? : (Listof Any) -> Boolean)
;; Tests whether a list is unique, guards Bind and Fun values.
(define (unique-list? xs)
  (or (null? xs)
      (and (not (member (first xs) (rest xs)))
           (unique-list? (rest xs)))))

(: parse-sexpr : Sexpr -> SLOTH)
;; parses s-expressions into SLOTHs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'bind more)
     (match sexpr
       [(list 'bind (list (list (symbol: names) (sexpr: nameds))
                          ...)
          body)
        (if (unique-list? names)
          (Bind names (map parse-sexpr nameds) (parse-sexpr body))
          (error 'parse-sexpr "duplicate `bind' names: ~s" names))]
       [else (error 'parse-sexpr "bad `bind' syntax in ~s" sexpr)])]
    [(cons 'fun more)
     (match sexpr
       [(list 'fun (list (symbol: names) ...) body)
        (if (unique-list? names)
          (Fun names (parse-sexpr body))
          (error 'parse-sexpr "duplicate `fun' names: ~s" names))]
       [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(cons 'if more)
     (match sexpr
       [(list 'if cond then else)
```

```
         (If (parse-sexpr cond)
             (parse-sexpr then)
             (parse-sexpr else))]
        [else (error 'parse-sexpr "bad `if' syntax in ~s" sexpr)])]
    [(list fun args ...) ; other lists are applications
     (Call (parse-sexpr fun)
           (map parse-sexpr args))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

(: parse : String -> SLOTH)
;; Parses a string containing an SLOTH expression to a SLOTH AST.
(define (parse str)
  (parse-sexpr (string->sexpr str)))

;;; ---------------------------------------------------------------
;;; Values and environments

(define-type ENV
  [EmptyEnv]
  [FrameEnv FRAME ENV])

;; a frame is an association list of names and values.
(define-type FRAME = (Listof (List Symbol VAL)))

(define-type VAL
  [RktV  Any]
  [FunV  (Listof Symbol) SLOTH ENV]
  [ExprV SLOTH ENV (Boxof (U #f VAL))]
  [PrimV ((Listof VAL) -> VAL)])

(: extend : (Listof Symbol) (Listof VAL) ENV -> ENV)
;; extends an environment with a new frame.
(define (extend names values env)
  (if (= (length names) (length values))
    (FrameEnv (map (lambda ([name : Symbol] [val : VAL])
                     (list name val))
                   names values)
              env)
    (error 'extend "arity mismatch for names: ~s" names)))

(: lookup : Symbol ENV -> VAL)
;; lookup a symbol in an environment, frame by frame,
;; return its value or throw an error if it isn't bound
(define (lookup name env)
  (cases env
    [(EmptyEnv) (error 'lookup "no binding for ~s" name)]
    [(FrameEnv frame rest)
     (let ([cell (assq name frame)])
       (if cell
         (second cell)
         (lookup name rest)))]))

(: unwrap-rktv : VAL -> Any)
;; helper for `racket-func->prim-val': strict and unwrap a RktV
;; wrapper in preparation to be sent to the primitive function
(define (unwrap-rktv x)
  (let ([s (strict x)])
    (cases s
```

```
      [(RktV v) v]
      [else (error 'racket-func "bad input: ~s" s)])))

(: wrap-in-val : Any -> VAL)
;; helper that ensures a VAL output using RktV wrapper when needed,
;; but leaving as is otherwise
(define (wrap-in-val x)
  (if (VAL? x) x (RktV x)))

(: racket-func->prim-val : Function Boolean -> VAL)
;; converts a racket function to a primitive evaluator function
;; which is a PrimV holding a ((Listof VAL) -> VAL) function.
;; (the resulting function will use the list function as is,
;; and it is the list function's responsibility to throw an error
;; if it's given a bad number of arguments or bad input types.)
(define (racket-func->prim-val racket-func strict?)
  (define list-func (make-untyped-list-function racket-func))
  (PrimV (lambda (args)
           (let ([args (if strict? (map unwrap-rktv args) args)])
             (wrap-in-val (list-func args))))))

;; The global environment has a few primitives:
(: global-environment : ENV)
(define global-environment
  (FrameEnv (list (list '+ (racket-func->prim-val + #t))
                  (list '- (racket-func->prim-val - #t))
                  (list '* (racket-func->prim-val * #t))
                  (list '/ (racket-func->prim-val / #t))
                  (list '< (racket-func->prim-val < #t))
                  (list '> (racket-func->prim-val > #t))
                  (list '= (racket-func->prim-val = #t))
                  ;; note flags:
                  (list 'cons  (racket-func->prim-val cons  #f))
                  (list 'list  (racket-func->prim-val list  #f))
                  (list 'first (racket-func->prim-val car   #t))
                  (list 'rest  (racket-func->prim-val cdr   #t))
                  (list 'null? (racket-func->prim-val null? #t))
                  ;; values
                  (list 'true  (RktV #t))
                  (list 'false (RktV #f))
                  (list 'null  (RktV null)))
            (EmptyEnv)))

;;; ----------------------------------------------------------------
;;; Evaluation

(: eval-promise : SLOTH ENV -> VAL)
;; used instead of `eval' to create an evaluation promise
(define (eval-promise expr env)
  (ExprV expr env (box #f)))

(: strict : VAL -> VAL)
;; forces a (possibly nested) ExprV promise, returns a VAL that is
;; not an ExprV
(define (strict val)
  (cases val
    [(ExprV expr env cache)
     (or (unbox cache)
```

```
         (let ([val* (strict (eval expr env))])
           (set-box! cache val*)
           val*))]
    [else val]))

(: eval : SLOTH ENV -> VAL)
;; evaluates SLOTH expressions
(define (eval expr env)
  ;; convenient helper
  (: eval* : SLOTH -> VAL)
  (define (eval* expr) (eval-promise expr env))
  (cases expr
    [(Num n)   (RktV n)]
    [(Id name) (lookup name env)]
    [(Bind names exprs bound-body)
     (eval bound-body (extend names (map eval* exprs) env))]
    [(Fun names bound-body)
     (FunV names bound-body env)]
    [(Call fun-expr arg-exprs)
     (define fval (strict (eval* fun-expr)))
     (define arg-vals (map eval* arg-exprs))
     (cases fval
       [(PrimV proc) (proc arg-vals)]
       [(FunV names body fun-env)
        (eval body (extend names arg-vals fun-env))]
       [else (error 'eval "function call with a non-function: ~s"
                    fval)])]
    [(If cond-expr then-expr else-expr)
     (eval* (if (cases (strict (eval* cond-expr))
                  [(RktV v) v] ; Racket value => use as boolean
                  [else #t])   ; other values are always true
                then-expr
                else-expr))]))

(: run : String -> Any)
;; evaluate a SLOTH program contained in a string
(define (run str)
  (let ([result (strict (eval (parse str) global-environment))])
    (cases result
      [(RktV v) v]
      [else (error 'run "evaluation returned a bad value: ~s"
                   result)])))

;;; ---------------------------------------------------------------
;;; Tests

(test (run "{{fun {x} {+ x 1}} 4}")
      => 5)
(test (run "{bind {{add3 {fun {x} {+ x 3}}}} {add3 1}}")
      => 4)
(test (run "{bind {{add3 {fun {x} {+ x 3}}}
                  {add1 {fun {x} {+ x 1}}}}
                {bind {{x 3}} {add1 {add3 x}}}}")
      => 7)
(test (run "{bind {{identity {fun {x} x}}
                  {foo {fun {x} {+ x 1}}}}
                {{identity foo} 123}}")
      => 124)
```

```
(test (run "{bind {{x 3}}
              {bind {{f {fun {y} {+ x y}}}}
                {bind {{x 5}}
                  {f 4}}}}")
      => 7)
(test (run "{{{fun {x} {x 1}}
              {fun {x} {fun {y} {+ x y}}}}
             123}")
      => 124)

;; More tests for complete coverage
(test (run "{bind x 5 x}")      =error> "bad `bind' syntax")
(test (run "{fun x x}")         =error> "bad `fun' syntax")
(test (run "{if x}")            =error> "bad `if' syntax")
(test (run "{}")                =error> "bad syntax")
(test (run "{bind {{x 5} {x 5}} x}") =error> "duplicate*bind*names")
(test (run "{fun {x x} x}")     =error> "duplicate*fun*names")
(test (run "{+ x 1}")           =error> "no binding for")
(test (run "{+ 1 {fun {x} x}}") =error> "bad input")
(test (run "{+ 1 {fun {x} x}}") =error> "bad input")
(test (run "{1 2}")             =error> "with a non-function")
(test (run "{{fun {x} x}}")     =error> "arity mismatch")
(test (run "{if {< 4 5} 6 7}")  => 6)
(test (run "{if {< 5 4} 6 7}")  => 7)
(test (run "{if + 6 7}")        => 6)
(test (run "{fun {x} x}")       =error> "returned a bad value")

;; Test laziness
(test (run "{{fun {x} 1} {/ 9 0}}") => 1)
(test (run "{{fun {x} 1} {{fun {x} {x x}} {fun {x} {x x}}}}") => 1)
(test (run "{bind {{x {{fun {x} {x x}} {fun {x} {x x}}}}} 1}") => 1)

;; Test lazy constructors
(test (run "{bind {{l {list 1 {/ 9 0} 3}}}
              {+ {first l} {first {rest {rest l}}}}}")
      => 4)

;;; ---------------------------------------------------------------
```

Side Effects in a Lazy Language

We've seen that a lazy language without the call-by-need optimization is too slow to be practical, but the optimization makes using side-effects extremely confusing. Specifically, when we deal with side-effects (I/O, mutation, errors, etc) the order of evaluation matters, but in our interpreter expressions are getting evaluated as needed. (Remember tracing the prime-numbers code in Lazy Racket — numbers are tested as needed, not in order.) If we can't do these things, the question is whether there is any point in using a purely functional lazy language at all — since computer programs often interact with an imperative world.

There is a solution for this: the lazy language does not have any (sane) facilities for *doing* things (like `printf` that prints something in plain Racket), but it can use a data structure that *describes* such operations. For example, in Lazy Racket we cannot print stuff sanely using `printf`, but we can construct a string using `format` (which is just like `printf`, except that it returns the formatted string instead of printing it). So (assuming Racket syntax for simplicity), instead of:

```
(define (foo n)
  (printf "~s + 1 = ~s\n" n (+ n 1)))
```

we will write:

```
(define (foo n)
  (format "~s + 1 = ~s\n" n (+ n 1)))
```

and get back a string. We can now change the way that our interpreter deals with the output value that it receives after evaluating a lazy expression: if it receives a string, then it can take that string as denoting a request for printout, and simply print it. Such an evaluator will do the printout when the lazy evaluation is done, and everything works fine because we don't try to use any side-effects in the lazy language — we just describe the desired side-effects, and constructing such a description does not require *performing* side-effects.

But this only solves printing a single string, and nothing else. If we want to print two strings, then the only thing we can do is concatenate the two strings — but that is not only inefficient, it cannot describe infinite output (since we will not be able to construct the infinite string in memory). So we need a better way to chain several printout representations. One way to do so is to use a list of strings, but to make things a little easier to manage, we will create a type for I/O descriptions — and populate it with one variant holding a string (for plain printout) and one for holding a chain of two descriptions (which can be used to construct an arbitrarily long sequence of descriptions):

```
(define-type IO
  [Print  String]
  [Begin2 IO IO])
```

Now we can use this to chain any number of printout representations by turning them into a single `Begin2` request, which is very similar to simply using a loop to print the list. For example, the eager printout code:

```
(: print-list : (Listof A) -> Void)
(define (print-list l)
  (if (null? l)
    (printf "\n")
    (begin (printf "~s " (first l))
           (print-list (rest l)))))
```

turns to the following code:

```
(: print-list : (Listof A) -> IO)
(define (print-list l)
  (if (null? l)
    (Print "\n")
    (Begin2 (Print (format "~s " (first l)))
            (print-list (rest l)))))
```

This will basically scan an input list like the eager version, but instead of printing the list, it will convert it into a single output request that forms a recipe for this printout. Note that within the lazy world, the result of `print-list` is just a value, there are no side effects involved. Turning this value into the actual printout is something that needs to be done on the eager side, which must be part of the implementation. In the case of Lazy Racket, we have no access to the implementation, but we can do so in our Sloth implementation: again, `run` will inspect the result and either print a given string (if it gets a `Print` value), or print two things recursively (if it gets a `Begin2` value). (To implement this, we will add an `IOV` variant to the `VAL` type definition, and have it contain an `IO` description of the above type.)

Because the sequence is constructed in the lazy world, it will not require allocating the whole sequence in memory — it can be forced bits by bits (using `strict`) as the imperative back-end (the `run` part of the implementation) follows the instructions in the resulting IO description. More concretely, it will also work on an infinite list: the translation of an infinite-loop printout function will be one that returns an infinite IO description tree of `Begin2` values. This loop will also force only what it needs to print and will go on recursively printing the whole sequence (possibly not terminating). For example (again, using Racket syntax), the infinite printout loop

```
(: print-loop : -> Void)
(define (print-loop)
```

```
  (printf "foo\n")
  (print-loop))
```

is translated into a function that returns an infinite tree of print operations:

```
(: print-loop : -> IO)
(define (print-loop)
  (Begin2 (Print "foo\n")
          (print-loop)))
```

When this tree is converted to actions, it will result in an infinite loop that produces the same output — it is essentially the same infinite loop, only now it's derived by an infinite description rather than an infinite process.

Finally, how should we deal with inputs? We can add another variant to our type definition that represents a `read-line` operation, assuming that like `read-line` it does not require any arguments:

```
(define-type IO
  [Print    String]
  [ReadLine ]
  [Begin2   IO IO])
```

Now the eager implementation can invoke `read-line` when it encounters a `ReadLine` value — but what should it do with the resulting string? Even worse, naively binding a value to `ReadLine`

```
(let ([name (ReadLine)])
  (Print (format "Your name is ~a" name)))
```

doesn't get us the string that is read — instead, the value is a *description* of a read operation, which is very different from the actual string value that we want in the binding.

The solution is to take the "code that acts on the string value" and make *it* be the argument to `ReadLine`. In the above example, that could would be the `let` expression without the `(ReadLine)` part — and as you rememebr from the time we introduced `fun` into `WAE`, taking away a named expression from a binding expression leads to a function. With this in mind, it makes sense to make `ReadLine` take a function value that represents what to do in the future, once the reading is actually done.

```
(ReadLine (lambda (name)
            (Print (format "Your name is ~a" name))))
```

This receiver value is a kind of a *continuation* of the computation, provided as a callback value — it will get the string that was read on the terminal, and will return a new description of side-effects that represents the rest of the process:

```
(define-type IO
  [Print    String]
  [ReadLine (String -> IO)]
  [Begin2   IO IO])
```

Now, when the eager side sees a `ReadLine` value, it will read a line, and invoke the callback function with the string that it has read. By doing this, the control goes back to the lazy world to process the value and get back another IO value to continue the processing. This results in a process where the lazy code generates some IO descriptions, then the imperative side will execute it and control goes back to the lazy code, then back to the imperative side, etc.

As a more verbose example of all of the above, this silly loop:

```
(: silly-loop : -> Void)
(define (silly-loop)
  (printf "What is your name? ")
  (let ([name (read-line)])
    (if (equal? name "quit")
      (printf "bye\n")
```

```
          (begin (printf "Your name is ~s\n" name)
                 (silly-loop)))))
```

is now translated to:

```
(: silly-loop : -> IO)
(define (silly-loop)
  (Begin2 (Print "What is your name? ")
          (ReadLine
           (lambda (name)
             (if (equal? name "quit")
               (Print "bye\n")
               (Begin2 (Print (format "Your name is ~s\n" name))
                       (silly-loop)))))))
```

Using this strategy to implement side-effects is possible, and you will do that in the homework — some technical details are going to be different but the principle is the same as discussed above. The last problem is that the above code is difficult to work with — in the homework you will see how to use syntactic abstractions to make things much simpler.