PL: Lecture #17

Tuesday, November 5th

Compilation

Instead of interpreting an expression, which is performing a full evaluation, we can think about *compiling* it: translating it to a different language which we can later run more easily, more efficiently, on more platforms, etc. Another feature that is usually associated with compilation is that a lot more work was done at the compilation stage, making the actual running of the code faster.

For example, translating an AST into one that has de-Bruijn indexes instead of identifier names is a form of compilation — not only is it translating one language into another, it does the work involved in name lookup before the program starts running.

This is something that we can experiment with now. An easy way to achieve this is to start with our evaluation function:

```
(: eval : TOY ENV -> VAL)
;; evaluates TOY expressions
(define (eval expr env)
  ;; convenient helper
  (: eval* : TOY -> VAL)
  (define (eval* expr) (eval 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 (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 (eval* cond-expr)
                  [(RktV v) v] ; Racket value => use as boolean
                  [else #t])   ; other values are always true
                then-expr
                else-expr))]))
```

and change it so it compiles a given expression to a Racket function. (This is, of course, just to demonstrate a conceptual point, it is only the tip of what compilers actually do…) This means that we need to turn it into a function that receives a TOY expression and compiles it. In other words, `eval` no longer consumes and environment argument which makes sense because the environment is a place to hold run-time values, so it is a data structure that is not part of the compiler (it is usually represented as the call stack).

So we split the two arguments into a compile-time and run-time, which can be done by simply currying the `eval` function — here this is done, and all calls to `eval` are also curried:

```
(: eval : TOY -> ENV -> VAL) ;*** note the curried type
;; evaluates TOY expressions
(define (eval expr)
  (lambda (env)
    ;; convenient helper
    (: eval* : TOY -> VAL)
    (define (eval* expr) ((eval 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 (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 (eval* cond-expr)
                    [(RktV v) v] ; Racket value => use as boolean
                    [else #t])   ; other values are always true
                  then-expr
                  else-expr))])))
```

We also need to change the `eval` call in the main `run` function:

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

Not much has changed so far.

Note that in the general case of a compiler we need to run a program several times, so we'd want to avoid parsing it over and over again. We can do that by keeping a single parsed AST of the input. Now we went one step further by making it possible to do more work ahead and keep the result of the first "stage" of eval around (except that "more work" is really not saying much at the moment):

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

At this point, even though our “compiler” is not much more than a slightly different representation of the same functionality, we rename `eval` to `compile` which is a more appropriate description of what we intend it to do (so we change the purpose statement too):

```
(: compile : TOY -> ENV -> VAL)
;; compiles TOY expressions to Racket functions.
(define (compile expr)
  (lambda (env)
    (: compile* : TOY -> VAL)
    (define (compile* expr) ((compile expr) env))
    (cases expr
      [(Num n)   (RktV n)]
      [(Id name) (lookup name env)]
      [(Bind names exprs bound-body)
       ((compile bound-body)
        (extend names (map compile* exprs) env))]
      [(Fun names bound-body)
       (FunV names bound-body env)]
      [(Call fun-expr arg-exprs)
       (define fval (compile* fun-expr))
       (define arg-vals (map compile* arg-exprs))
       (cases fval
         [(PrimV proc) (proc arg-vals)]
         [(FunV names body fun-env)
          ((compile body) (extend names arg-vals fun-env))]
         [else (error 'call ; this is *not* a compilation error
                      "function call with a non-function: ~s"
                      fval)])]
      [(If cond-expr then-expr else-expr)
       (compile* (if (cases (compile* 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 TOY program contained in a string
(define (run str)
  (let* ([compiled (compile (parse str))]
         [result   (compiled global-environment)])
    (cases result
      [(RktV v) v]
      [else (error 'run "evaluation returned a bad value: ~s"
                   result)])))
```

Not much changed, still. We curried the `eval` function and renamed it to `compile`. But when we actually call compile almost nothing happens — all it does is create a Racket closure which will do the rest of the work. (And this closure closes over the given expression.)

Running this “compiled” code is going to be very much like the previous usage of `eval`, except a little *slower*, because now every recursive call involves calling `compile` to generate a closure, which is then immediately used — so we just added some allocations at the recursive call points! (Actually, the extra cost is minimal because the Racket compiler will optimize away such immediate closure applications.)

Another way to see how this is not really a compiler yet is to consider *when* `compile` gets called. A proper compiler is something that does all of its work *before* running the code, which means that once it spits out the compiled code it shouldn’t be used again (except for compiling other code, of course). Our current code is not really a compiler since it breaks this feature. (For example, if GCC behaved this way, then it would

“compile” files by producing code that invokes GCC to compile the next step, which, when run, invokes GCC again, etc.)

However, the conceptual change is substantial — we now have a function that does its work in two stages — the first part gets an expression and *can* do some compile-time work, and the second part does the run-time work, and includes anything inside the (lambda (env) …). The thing is that so far, the code does nothing at the compilation stage (remember: only creates a closure). But because we have two stages, we can now shift work from the second stage (the run-time) to the first (the compile-time).

For example, consider the following simple example:

```
#lang pl

(: foo : Number Number -> Number)
(define (foo x y)
  (* x y))

(: bar : Number -> Number)
(define (bar c)
  (: loop : Number Number -> Number)
  (define (loop n acc)
    (if (< 0 n)
        (loop (- n 1) (+ (foo c n) acc))
        acc))
  (loop 4000000000 0))

(time (bar 0))
```

We can do the same thing here — separate `foo` it into two stages using currying, and modify `bar` appropriately:

```
#lang pl

(: foo : Number -> Number -> Number)
(define (foo x)
  (lambda (y)
    (* x y)))

(: bar : Number -> Number)
(define (bar c)
  (: loop : Number Number -> Number)
  (define (loop n acc)
    (if (< 0 n)
        (loop (- n 1) (+ ((foo c) n) acc))
        acc))
  (loop 4000000000 0))

(time (bar 0))
```

Now instead of a simple multiplication, lets expand it a little, for example, do a case split on common cases where `x` is `0`, `1`, or `2`:

```
(: foo : Number -> Number -> Number)
(define (foo x)
  (lambda (y)
    (cond [(= x 0) 0]
          [(= x 1) y]
          [(= x 2) (+ y y)] ; assume that this is faster
          [else (* x y)])))
```

This is not much faster, since Racket already optimizes multiplication in a similar way.

Now comes the real magic: deciding what branch of the cond to take depends *only* on x, so we can `push` the lambda inside:

```
(: foo : Number -> Number -> Number)
(define (foo x)
  (cond [(= x 0) (lambda (y) 0)]
        [(= x 1) (lambda (y) y)]
        [(= x 2) (lambda (y) (+ y y))]
        [else (lambda (y) (* x y))]))
```

We just made an improvement — the comparisons for the common cases are now done as soon as (foo x) is called, they're not delayed to when the resulting function is used. Now go back to the way this is used in `bar` and make it call `foo` once for the given `c`:

```
#lang pl

(: foo : Number -> Number -> Number)
(define (foo x)
  (cond [(= x 0) (lambda (y) 0)]
        [(= x 1) (lambda (y) y)]
        [(= x 2) (lambda (y) (+ y y))]
        [else (lambda (y) (* x y))]))

(: bar : Number -> Number)
(define (bar c)
  (define foo-c (foo c))
  (: loop : Number Number -> Number)
  (define (loop n acc)
    (if (< 0 n)
        (loop (- n 1) (+ (foo-c n) acc))
        acc))
  (loop 4000000000 0))

(time (bar 0))
```

Now foo-c is generated once, and if `c` happens to be one of the three common cases (as in the last expression), we can avoid doing any multiplication. (And if we hit the default case, then we're doing the same thing we did before.)

[However, the result runs at roughly the same speed! This heavily depends on what kind of optimizations the compiler can do, in this case, optimizing multiplications (which are essentially a single machine-code instruction) vs optimizing multiple-stage function calls.]

Here is another useful example that demonstrates this:

```
(define (foo list)
  (map (lambda (n) (if ...something... E1 E2))
       list))

-->

(define (foo list)
  (map (if ...something...
           (lambda (n) E1)
           (lambda (n) E2))
       list))
```

(Question: when can you do that?)

This is not unique to Racket, it can happen in any language. Racket (or any language with first class function values) only makes it easy to create a local function that is specialized for the flag.

Getting our thing closer to a compiler is done in a similar way — we push the `(lambda (env) ...)` inside the various cases. (Note that `compile*` depends on the `env` argument, so it also needs to move inside — this is done for all cases that use it, and will eventually go away.) We actually need to use `(lambda ([env : ENV]) ...)` though, to avoid upsetting the type checker:

```
(: compile : TOY -> ENV -> VAL)
;; compiles TOY expressions to Racket functions.
(define (compile expr)
  (cases expr
    [(Num n)   (lambda ([env : ENV]) (RktV n))]
    [(Id name) (lambda ([env : ENV]) (lookup name env))]
    [(Bind names exprs bound-body)
     (lambda ([env : ENV])
       (: compile* : TOY -> VAL)
       (define (compile* expr) ((compile expr) env))
       ((compile bound-body)
        (extend names (map compile* exprs) env)))]
    [(Fun names bound-body)
     (lambda ([env : ENV]) (FunV names bound-body env))]
    [(Call fun-expr arg-exprs)
     (lambda ([env : ENV])
       (: compile* : TOY -> VAL)
       (define (compile* expr) ((compile expr) env))
       (define fval (compile* fun-expr))
       (define arg-vals (map compile* arg-exprs))
       (cases fval
         [(PrimV proc) (proc arg-vals)]
         [(FunV names body fun-env)
          ((compile body) (extend names arg-vals fun-env))]
         [else (error 'call ; this is *not* a compilation error
                      "function call with a non-function: ~s"
                      fval)]))]
    [(If cond-expr then-expr else-expr)
     (lambda ([env : ENV])
       (: compile* : TOY -> VAL)
       (define (compile* expr) ((compile expr) env))
       (compile* (if (cases (compile* cond-expr)
                       [(RktV v) v] ; Racket value => use as boolean
                       [else #t])   ; other values are always true
                     then-expr
                     else-expr)))]))
```

and with this we shifted a bit of actual work to compile time — the code that checks what structure we have, and extracts its different slots. But this is still not good enough — it's only the first top-level `cases` that is moved to compile-time — recursive calls to `compile` are still there in the resulting closures. This can be seen by the fact that we have those calls to `compile` in the Racket closures that are the results of our compiler, which, as discussed above, mean that it's not an actual compiler yet.

For example, consider the `Bind` case:

```
[(Bind names exprs bound-body)
 (lambda ([env : ENV])
   (: compile* : TOY -> VAL)
   (define (compile* expr) ((compile expr) env))
   ((compile bound-body)
    (extend names (map compile* exprs) env)))]
```

At compile-time we identify and deconstruct the Bind structure, then create a the runtime closure that will access these parts when the code runs. But this closure will itself call `compile` on `bound-body` and each

of the expressions. Both of these calls can be done at compile time, since they only need the expressions — they don't depend on the environment. Note that `compile*` turns to `run` here, since all it does is run a compiled expression on the current environment.

```
[(Bind names exprs bound-body)
 (define compiled-body (compile bound-body))
 (define compiled-exprs (map compile exprs))
 (lambda ([env : ENV])
   (: run : (ENV -> VAL) -> VAL)
   (define (run compiled-expr) (compiled-expr env))
   (compiled-body (extend names
                          (map run compiled-exprs)
                          env)))]
```

We can move it back up, out of the resulting functions, by making it a function that consumes an environment and returns a "caller" function:

```
(define (compile expr)
  ;; convenient helper
  (: caller : ENV -> (ENV -> VAL) -> VAL)
  (define (caller env)
    (lambda (compiled) (compiled env)))
  (cases expr
    ...
    [(Bind names exprs bound-body)
     (define compiled-body (compile bound-body))
     (define compiled-exprs (map compile exprs))
     (lambda ([env : ENV])
       (compiled-body (extend names
                              (map (caller env) compiled-exprs)
                              env)))]
    ...))
```

Once this is done, we have a bunch of work that can happen at compile time: we pre-scan the main "bind spine" of the code.

We can deal in a similar way with other occurrences of `compile` calls in compiled code. The two branches that need to be fixed are:

1. In the `If` branch, there is not much to do. After we make it pre-compile the `cond-expr`, we also need to make it pre-compile both the `then-expr` and the `else-expr`. This might seem like doing more work (since before changing it only one would get compiled), but since this is compile-time work, then it's not as important. Also, `if` expressions are evaluated many times (being part of a loop, for example), so overall we still win.
2. The `Call` branch is a little trickier: the problem here is that the expressions that are compiled are coming from the closure that is being applied. The solution for this is obvious: we need to change the closure type so that it closes over *compiled* expressions instead of over plain ones. This makes sense because closures are run-time values, so they need to close over the compiled expressions since this is what we use as "code" at run-time.

Again, the goal is to have no `compile` calls that happen at runtime: they should all happen before that. This would allow, for example, to obliterate the compiler once it has done its work, similar to how you don't need GCC to run a C application. Yet another way to look at this is that we shouldn't look at the AST at runtime — again, the analogy to GCC is that the AST is a data structure that the compiler uses, and it does not exist at runtime. Any runtime reference to the TOY AST is, therefore, as bad as any runtime reference to `compile`.

When we're done with this process we'll have something that is an actual compiler: translating TOY programs into Racket closures. To see how this is an actual compiler consider the fact that Racket uses a JIT to translate bytecode into machine code when it's running functions. This means that the compiled version of our TOY programs are, in fact, translated all the way down to machine code.

Is this really a compiler?

Yes, it is, though it's hard to see it when we're compiling TOY code directly to Racket closures.

Another way to see this in a more obvious way is to change the compiler code so instead of producing a Racket closure it spits out the Racket code that makes up these closures when evaluated.

The basic idea is to switch from a function that has code that "does stuff", to a function that *emits* that code indtead. For example, consider a function that computes the average of two numbers

```
(define (average x y)
  (/ (+ x y) 2))
```

to one that instead returns the actual code

```
(define (make-average-expression x y)
  (string-append "(/ (+ " x " " y ") 2)"))
```

It is, however, inconvenient to use strings to represent code: S-expressions are a much better fit for representing code:

```
(define (make-average-expression x y)
  (list '/ (list '+ x y) 2))
```

This is still tedious though, since the clutter of `list`s and quotes makes it hard to see the actual code that is produced. It would be nice if we could quote the whole thing instead:

```
(define (make-average-expression x y)
  '(/ (+ x y) 2))
```

but that's wrong since we don't want to include the `x` and `y` symbols in the result, but rather their values. Racket (and all other lisp dialects) have a tool for that: `quasiquote`. In code, you just use a backquote `` ` `` instead of a `'`, and then you can unquote parts of the quasi-quoted code using `,` (which is called `unquote`). (Later in the course we'll talk about these "`` ` ``"s and "`,`"s more.)

So the above becomes:

```
(define (make-average-expression x y)
  `(/ (+ ,x ,y) 2))
```

Note that this would work fine if `x` and `y` are numbers, but they're now essentially arguments that hold *expression* values (as S-expressions). For example, see what you get with:

```
(make-average-expression 3 `(/ 8 2))
```

Back to the compiler, we change the closure-generating compiler code

```
[(Bind names exprs bound-body)
 (define compiled-body (compile bound-body))
 (define compiled-exprs (map compile exprs))
 (lambda ([env : ENV])
   (compiled-body (extend ...)))]
```

into

```
[(Bind names exprs bound-body)
 (define compiled-body (compile bound-body))
 (define compiled-exprs (map compile exprs))
 `(lambda ([env : ENV])
    (,compiled-body (extend ...)))]
```

Doing this systematically would result in something that is more clearly a compiler: the result of `compile` would be an S-expression that you can then paste in the Racket file to run it.

An example of this idea taken seriously is the graal + truffle combination for implementing fast JIT compiled languages:

- **https://medium.com/graalvm/writing-truly-memory-safe-jit-compilers-f79ad44558dd**