Assignment 13

Contents

1. Background

In this assignment, we generalize our language to allow `lambda` expressions to appear anywhere other expressions can appear, i.e., not just as the right-hand side of a `letrec` binding.

We also add a couple of optimization passes, one that converts direct `lambda` applications into `let` expressions and one that replaces indirect calls with direct calls to labels at call sites where the callee is known.

2. Scheme Subset 13

Here is a grammar for the subset of Scheme we'll be handling this week:

```
Program    —→  Expr
Expr       —→  uvar
            |  (quote Immediate)
            |  (if Expr Expr Expr)
            |  (begin Expr* Expr)
            |  Lambda
            |  (let ([uvar Expr]*) Expr)
            |  (letrec ([uvar Lambda]*) Expr)
            |  (prim Expr*)
            |  (Expr Expr*)
Lambda     —→  (lambda (uvar*) Expr)
Immediate  —→  fixnum | () | #t | #f
```

The set of primitives and the constraints on unique variables and fixnums do not change.

3. Things to do

In the source language of Assignment 12, a `lambda` expression can appear only on the right-hand side of a `letrec` binding, so this week's passes must arrange for this to be the case. We do this in two steps, first giving each anonymous `lambda` a name (`remove-anonymous-lambda`) and then converting `let` bindings whose right-hand sides are `lambda` expressions into `letrec` bindings (`sanitize-binding-forms`).

We perform direct-call and known-call optimization in `optimize-direct-call` and `optimize-known-call`.

3.1. verify-scheme

This pass must be updated to reflect the change in the language, i.e., to allow `lambda` expressions to appear anywhere other expressions can appear.

3.2. optimize-direct-call

Before eliminating anonymous `lambda` expressions, we have an opportunity to optimize those that appear in the operator position of an application. These direct `lambda` applications are often introduced by a macro expander that translates programs, prior to compilation, from a rich source language into the simpler core handled by the compiler proper. Direct calls may also arise from earlier compiler transformations.

For example, `let` expressions are often translated by a macro expander as follows.

```
(let ((x e) ...) body)  →  ((lambda (x ...) body) e ...)
```

This pass performs the opposite transformation. For example,

```
((lambda (x.1) x.1) '3)  →  (let ([x.1 '3]) x.1)
```

This replaces what would otherwise be an anonymous call to an anonymous `lambda` with a simple `let` expression. In practical terms, this transformation avoids an unecessary heap allocation, an indirect jump, and, when there are free variables in the body, some additional indirect memory references. If the expression is evaluated frequently, the savings from this nearly trivial optimization can be significant.

Performing this optimization is a simple matter of recognizing the direct application pattern and converting it to `let`. It should proceed only if the number of formal parameters of the `lambda` expressions matches the number of actual parameters of the enclosing call.

3.3. remove-anonymous-lambda

This pass introduces a unique name for each `lambda` expression that does not appear as the right-hand side of a `let` or `letrec` expression. It does this via the following transformation.

```
(lambda (formal ...) body)  →
  (letrec ([anon (lambda (formal ...) body)])
    anon)
```

where *anon* is a new unique variable.

For example:

```
(let ([f.3 (lambda (x.1)
             (lambda (y.2)
               (+ x.1 y.2)))])
  ((f.3 '3) '8))
```

becomes

```
(let ([f.3 (lambda (x.1)
             (letrec ([anon.4 (lambda (y.2) (+ x.1 y.2))])
               anon.4))])
  ((f.3 '3) '8))
```

Although `optimize-direct-call`, which runs before this pass, eliminates anonymous `lambda` expressions that appear in operator position, we do not rely on this here since `optimize-direct-call` is an optimization pass and might not be enabled.

3.4. `sanitize-binding-forms`

At this point, all `lambda` expressions have names, which is an important step toward the ultimate goal of representing the code of each procedure by a labeled code block. To produce a program in the language accepted by the rest of the compiler, however, the names must all be given by `letrec`, i.e., a `lambda` expression must appear only as the right-hand side of a `letrec` binding and never as the right-hand side of a `let` binding.

To bring this about, we partition the bindings of each `let` expression into the set that bind `lambda` expressions and the set that do not. We lift those that bind `lambda` expressions out and place them in a `letrec` expression wrapped around what remains of the `let` expression. For example,

```
(let ([x.4 '0] [f.2 (lambda (x.1) x.1)] [y.3 '1])
  (+ x.4 (f.2 y.3)))
```

becomes

```
(letrec ([f.2 (lambda (x.1) x.1)])
  (let ([x.4 '0] [y.3 '1])
    (+ x.4 (f.2 y.3))))
```

This transformation would not be sound if variables were not uniquely named, because the lifted `letrec` bindings could improperly capture free references in the `letrec` and `let` right-hand sides. In addition, it would be unsound in the presence of `call/cc` and assignments, which could be used to expose the fact that only one location is created for each generated `letrec` binding rather than one for each time a continuation of a `let` right-hand side is invoked.

Putting the `letrec` expression within the `let` expression also works with our intermediate language.

```
(let ([x.4 '0] [y.3 '1])
  (letrec ([f.2 (lambda (x.1) x.1)])
    (+ x.4 (f.2 y.3))))
```

To avoid clutter in the output, this pass should avoid producing `let` and `letrec` expressions that bind no variables. That is, if a `let` or `letrec` expression that results from the transformation above binds no variables, it should be suppressed. Similarly, `let` and `letrec` expressions in the input of this pass might as well be eliminated if they do not bind any variables. Thus, this pass should perform the following simple transformations as it reconstitutes `let` and `letrec` bindings.

`(let ()` *Body*`)` → *Body*

`(letrec ()` *Body*`)` → *Body*

As with `make-begin`'s flattening of `begin` expressions (which is also useful in this pass), this is done for convenience only, and is not something we consider to be a change in the language. We do not assume nonempty binding lists in later passes, just as we do not assume flattened `begin` expressions.

The output language of this pass is the same as the input language for Assignment 12.

The transformation this pass performs is not sound in a source language that includes both `call/cc` and variable assignments. Even if we decide to add `call/cc`, it is safe to make the transformation at this point because we do not have variable assignments.

3.5. `optimize-known-call`

For each procedure call in the program, the operator is assumed to evaluate to a procedure, represented as a closure. The generated code for a call generally must extract the address of the closed procedure from the closure and jump to the extracted address. This is referred to as an *indirect call*: rather than calling the closed procedure directly, the code calls indirect through the closure. This is suitable for *anonymous calls*, calls in which the call site does not know exactly which procedure is being invoked. For calls to *known* procedures, however, it is more efficient to call the closed procedure directly.

The goal of `optimize-known-call` is thus to convert each anonymous call of the form:

(e_0 e_1 ...)

into

(*label* e_1 ...)

when the label of the `lambda` expression to which e_0 evaluates can be determined. The former requires both a memory indirect (made explicit via the use of `procedure-code` in `introduce-procedure-primitives`) and an indirect jump, both of which are costly, while the latter involves a direct (unconditional) jump.

This pass runs after `convert-closures` but before `introduce-procedure-primitives`, at which point each `letrec`-bound procedure is an explicitly labeled closed procedure that expects its first argument to be a closure containing the values of its (formerly) free variables. Each `closures`-bound procedure is a closure that encapsulates both a pointer to the corresponding closed procedure and the values of its free variables.

Although an elaborate analysis can be used to determine known procedures, for our purposes a procedure is known at a call site if and only if the procedure expression is a variable and the variable is bound by a `closures` form.

This pass therefore converts calls to `closures`-bound variables into direct calls to the corresponding closed procedures. If *f*`.`*n* is a `closures`-bound variable, and *f*`$`*n* is the label of the corresponding closed procedure, the call

(*f*`.`*n* e_1 ...)

becomes, simply

(*f*`$`*n* e_1 ...)

In the Scheme program below, the calls to `f` are anonymous, while the calls to `map` and `mulx` are to known procedures.

```
(letrec ([map (lambda (f ls)
                (if (null? ls)
                    '()
                    (cons (f (car ls))
                          (map f (cdr ls)))))])
  (let ([mulx (lambda (x)
                (lambda (y)
                  (* x y)))])
    (map (mulx 7)
         (map (lambda (z) (+ z 1))
              (cons 1 (cons 2 '()))))))
```

After converting this into our language by replacing the variables with unique variables and running all passes through `convert-closures`, the following program appears in the input to `optimize-known-call`:

```
(letrec ([map$1 (lambda (cp.13 f.3 ls.2)
                  (bind-free (cp.13 map.1)
                    (if (null? ls.2)
                        '()
                        (cons
                          (f.3 f.3 (car ls.2))
```

```
                         (map.1 map.1 f.3 (cdr ls.2))))))])
  (closures ([map.1 map$1 map.1])
    (letrec ([mulx$6 (lambda (cp.12 x.4)
                       (bind-free (cp.12)
                         (letrec ([anon$9 (lambda (cp.11 y.5)
                                            (bind-free (cp.11 x.4)
                                              (* x.4 y.5)))])
                           (closures ([anon.9 anon$9 x.4]) anon.9))))])
      (closures ([mulx.6 mulx$6])
        (map.1 map.1
          (mulx.6 mulx.6 '7)
          (map.1 map.1
            (letrec ([anon$8 (lambda (cp.10 z.7)
                               (bind-free (cp.10) (+ z.7 '1)))])
              (closures ([anon.8 anon$8]) anon.8))
            (cons '1 (cons '2 '()))))))))
```

and `optimize-known-call` produces the following.

```
(letrec ([map$1 (lambda (cp.13 f.3 ls.2)
                  (bind-free (cp.13 map.1)
                    (if (null? ls.2)
                        '()
                        (cons
                          (f.3 f.3 (car ls.2))
                          (map$1 map.1 f.3 (cdr ls.2))))))])
  (closures ([map.1 map$1 map.1])
    (letrec ([mulx$6 (lambda (cp.12 x.4)
                       (bind-free (cp.12)
                         (letrec ([anon$9 (lambda (cp.11 y.5)
                                            (bind-free (cp.11 x.4)
                                              (* x.4 y.5)))])
                           (closures ([anon.9 anon$9 x.4]) anon.9))))])
      (closures ([mulx.6 mulx$6])
        (map$1 map.1
          (mulx$6 mulx.6 '7)
          (map$1 map.1
            (letrec ([anon$8 (lambda (cp.10 z.7)
                               (bind-free (cp.10) (+ z.7 '1)))])
              (closures ([anon.8 anon$8]) anon.8))
            (cons '1 (cons '2 '()))))))))
```

The only changes are in the calls to `map.1` and `mulx.6`, in which the operator has been replaced by `map$1` and `mulx$6`.

Because this is an optimization pass, the input and output languages are the same. The pass might be disabled and the rest of the compiler must be able to produce correct code for either the unoptimized or the optimized form of the program.

In coding this pass, it is necessary to recognize the `letrec` and `closures` forms as a single composite expression so the optimization performed by this pass can be made both within the body of the `closures` form and within the right-hand side expressions of the `letrec`. Otherwise, the optimizer will miss some opportunities to apply the optimization. For example, if you do not look for and replace calls to `closures`-bound uvars in the right-hand side expressions of the corresponding `letrec`, the recursive calls to `map.1` within `map.1` in the program above will not be optimized, and much of the benefit will be lost.

4. Boilerplate and Run-time Code

The boilerplate code and run-time code do not change.

5. Testing

A small set of invalid and valid tests for this assignment have been posted in tests13.ss. You should make sure that your compiler passes work at least on this set of tests.

6. Coding Hints

Before starting, study the output of the online compiler for several examples.