Assignment 11

Contents

1. Background 1
2. Scheme Subset 11 1
3. Things to do 2
   3.1. `verify-scheme` 2
   3.2. `lift-letrec` 2
   3.3. `normalize-context` 2
   3.4. `optimize-jumps` 4
4. Boilerplate and Run-time Code 6
5. Testing 6
6. Coding Hints 6

1. Background

In this assignment, we remove restrictions on where primitive calls, procedure calls, and quoted constants appear and assign meaning to those that appear in "abnormal" contexts, such as predicate primitive calls in value context or value primitive calls in effect context. We also generalize the language to permit `letrec` expressions to appear anywhere, not just at top level, although we still do not permit `lambda` expressions to have free variables.

Finally, we add an optimization pass that eliminates the jumps to jumps that `expose-basic-blocks` produces in certain circumstances.

2. Scheme Subset 11

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

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

Within the same *Program*, each *label* bound by a `letrec` expression must have a unique suffix, and each *uvar* bound by a `lambda` or `let` expression must have a unique suffix.

A *label* is visible throughout the `letrec` expression that binds it. A *uvar* is visible throughout the `let` or `lambda` body that binds it, except within nested `lambda` expressions. Thus, a *label* can appear free within a `lambda` expression, but a *uvar* cannot.

A *fixnum* is an exact integer in a machine-dependent range, which can be determined from the helpers.ss `fixnum-bits`.

The set of primitives *prim* is the union of the sets of value, pred, and effect primitives given in Assignment 10.

3. Things to do

We need to rewrite the program so that it is in the source language of Assignment 10, which means we need to move all `letrec` expressions to top level and reorganize the code into separate *Value*, *Pred*, and *Effect* contexts. We achieve this through two new passes, `lift-letrec` and `normalize-context`. We must also add an optimization pass, `optimize-jumps`, that runs after `expose-basic-blocks` to eliminate the jumps to jumps that `expose-basic-blocks` can leave behind.

3.1. `verify-scheme`

This pass must be updated to reflect the changes in the language.

3.2. `lift-letrec`

This pass simply moves all `letrec` bindings from where they appear into a `letrec` expression wrapped around the outermost expression, removing all of the internal `letrec` expressions in the process.

3.3. `normalize-context`

On input to this pass, any expression can appear in any of three contexts:

effect, where the resulting value is not needed, e.g., all but the last subexpression of a `begin`;

predicate, where the expression determines flow of control, e.g., the test part of an `if`; and

value, where the value is needed, e.g., the right-hand side of a `let`.

Since any expression can appear in any of the three context, the contexts are not even distinguished by the grammar given in Section 2.

Yet, some expressions don't make sense in certain contexts, e.g., constants in effect context. Others are problematic as we move toward language-independent code. In the machine-independent portion of the compiler, the compiler cannot (should not!) know what value to return for `(< 3 4)` when it appears in a value context. Should it return `#t`? That wouldn't be language-independent. How about `1`, as in C? That wouldn't be language-independent either. Does `(- 17 17)` evaluate to a true or a false value? Which values returned by a procedure call are true and which are false? There's no way for the language-independent portion of the compiler to know.

The goal of this pass is to recognize the three distinct contexts above and to rewrite the code so that each kind of expression appears only in contexts that are appropriate for that expression.

Along with this partitioning of contexts goes a partitioning of primitives. Calls to predicate primitives, such as `<`, are permitted only in predicate context. Calls to value primitives, such as `+`, are permitted only in value context. Calls to effect primitives, such as `set-car!`, are permitted only in effect context. Each of our primitives falls into one of these categories.

In the input language, the Scheme primitive `void` can be used as both a value (meaning the unspecified value) and as a "no-op" (no operation) place-holder, e.g., in the representation of one-armed `if` expressions as two-armed `if` expressions. In the output of `normalize-context`, `(nop)` is used in place of a call to `void` for the latter purpose.

Besides primitive calls, a few other expressions are limited to certain contexts in the output of this pass

- Constants, i.e., `quote` expressions, are permitted only in value context. They are unnecessary in effect context and are simply dropped, i.e., replaced with `(nop)`, in that context. In predicate context, `#f` is replaced by `(false)`, and all nonfalse constants are replaced by `(true)`.

- Variable references and `label` expressions are valid only in value context.
- Procedure calls are valid in both value and effect context but not in predicate context.

An `if`, `begin`, or `let` expression may appear in any of the three contexts.

The partitioning into contexts and restrictions on what may be present in each context are reflected in the grammar for the output of this pass, which is the grammar for the source language of Assignment 10.

The following contrived input program shows `normalize-context` in action.

```
(letrec ()
  (let ([x.1 '1] [y.2 '2] [a.3 '3] [b.4 '4] [p.5 (cons '#f '#t)])
    (begin
      (* (begin (set-car! p.5 '#t) x.1) y.2)
      (if (if (car p.5) (if (< x.1 y.2) '#f '#t) '17)
          (if (= a.3 b.4) '#f '#t)
          (<= y.2 x.1)))))
```

Ths program contains several violations of the restrictions `normalize-context` is designed to enforce. While the call to `<` is used appropriately to direct traffic, determining which of its two arms should proceed, the call to `<=` is in value context. The call to `*` appears in effect context, where it is useless, as are the references to `x.1` and `y.2`. The call to `car` appears in a predicate context, but `car` is more appropriately treated as a value primitive. Finally, the expression `'17` appears in a predicate context, as do a couple of quoted boolean constants.

Here is what `normalize-context` produces for this program.

```
(letrec ()
  (let ([x.1 '1] [y.2 '2] [a.3 '3] [b.4 '4] [p.5 (cons '#f '#t)])
    (begin
      (set-car! p.5 '#t)
      (if (if (if (eq? (car p.5) '#f) (false) (true))
              (if (< x.1 y.2) (false) (true))
              (true))
          (if (= a.3 b.4) '#f '#t)
          (if (<= y.2 x.1) '#t '#f)))))
```

The call to `<=` has been replaced with an equivalent `if` expression that explicitly computes the boolean value, so that the resulting call is in predicate context. The call to `*` and the variable references to `x.1` and `y.1` have been eliminated. The call to `car` has been replaced with `(if (eq? (car p.5) '#f) (false) (true))`, explicitly testing its boolean value. Finally, the constants appearing in predicate context have been replaced with `(true)` or `(false)` as appropriate.

Each transformation is straightforward and generally applicable to similar kinds of expressions. Any constant in predicate context can be converted to `(true)` or `(false)`. Any other value-producing expression *expr* in predicate context can be converted to

```
(if (eq? expr '#f) (false) (true))
```

Any predicate expression *expr* in value context can be converted to

```
(if expr '#t '#f)
```

In effect context, constants, variables, and label references can be converted into `(nop)`, but care must be taken for value and predicate primitive calls. While the calls may be discarded, the operands must be evaluated for their effects, if any. This can be done by processing the operands in effect context and wrapping the resulting code in a `begin` expression. Unnecessary occurrences of `(nop)` should be removed using a helper `make-nopless-begin`.

```
(define (make-nopless-begin x*)
  (let ([x* (remove '(nop) x*)])
    (if (null? x*)
        '(nop)
        (make-begin x*))))
```

The set of input expressions is small, but we have to consider each type in each of the three contexts, and we have to recognize three different categories of primitives. For each context, we must look for each kind of expression and deal with it in a matter appropriate for the context so that the output language is in the grammar shown above. The structure of the pass must therefore look like many of the passes we have already written, with separate handlers for *Value*, *Pred*, and *Effect* contexts. In general, the high-level structure of a pass typically reflects the structure of the output grammar, although the set of expressions it must process is determined by the structure of the input grammar.

3.4. `optimize-jumps`

When we added `expose-basic-blocks` to our compilers, we decided not to special-case `if` expressions where the consequent or alternative is `(nop)`, `(true)`, or `(false)`, even though the pass would have generated better code had wse done so. We did this for two reasons: (1) to avoid complicating and already complicated pass and (2) because we generally want to do optimizations in passes dedicated to performing those optimizations. We put optimizations in dedicated passes so that they can be selectively disabled for testing and to allow programmers to trade run-time performance for compilation speed during program development.

In each case where we could have special-cased an `if` expression, `expose-basic-blocks` generates a `letrec` binding to a procedure that simply jumps to another block. While these extra blocks do not affect the correctness of the generated code, they do affect the size and possibly performance of the generated code. The new optimization pass, `optimize-jumps`, runs after `expose-basic-blocks` and eliminate these unnecessary jumps.

For instance, let's say we have the program:

```
(letrec ()
  (let ([x.1 '5])
    (if (if (= x.1 '6) (true) (if (= x.1 '5) (true) (= x.1 '4)))
        (+ x.1 x.1)
        (* x.1 2))))
```

`expose-basic-blocks` generates something like the following code for this program:

```
(letrec ([c$9 (lambda () (c$5))]
         [a$10 (lambda () (if (= rax 40) (c$7) (a$8)))]
         [c$7 (lambda () (c$5))]
         [a$8 (lambda () (if (= rax 32) (c$5) (a$6)))]
         [c$5 (lambda () (begin (set! rax (+ rax rax)) (r15)))]
         [a$6 (lambda () (begin (set! rax (* rax 2)) (r15)))])
  (begin (set! rax 40) (if (= rax 48) (c$5) (a$10))))
```

Two of the `lambda` bodies consist solely of jumps. After `optimize-jumps`, the bindings for these `lambda` bodies no longer appear.

```
(letrec ([a$10 (lambda () (if (= rax 40) (c$5) (a$8)))]
         [a$8 (lambda () (if (= rax 32) (c$5) (a$6)))]
         [c$5 (lambda () (begin (set! rax (+ rax rax)) (r15)))]
         [a$6 (lambda () (begin (set! rax (* rax 2)) (r15)))])
  (begin (set! rax 40) (if (= rax 48) (c$5) (a$10))))
```

In addition to removing the bindings, the pass also replaces the jumps to `c$7` and `c$9` with jumps to `c$5`, which happens to be the ultimate target in each case.

The pass can perform the optimization in three logical steps:

- build an association list from the label of each `lambda` expression whose body is a jump to its target label; and
- remove the bindings for the `lambda` expressions whose bodies are jumps; and
- within the remaining bodies, replace the target of each jump to one of these `lambda` expressions with a jump directly to the final target.

The first and second logical steps can be done at the same time, but we will need to build the association list before we can do the final step.

Walking through our example above:

```
(letrec ([c$9 (lambda () (c$5))]
         [a$10 (lambda () (if (= rax 40) (c$7) (a$8)))]
         [c$7 (lambda () (c$5))]
         [a$8 (lambda () (if (= rax 32) (c$5) (a$6)))]
         [c$5 (lambda () (begin (set! rax (+ rax rax)) (r15)))]
         [a$6 (lambda () (begin (set! rax (* rax 2)) (r15)))])
  (begin (set! rax 40) (if (= rax 48) (c$5) (a$10))))
```

Performing both the first and second steps, we eliminate the `letrec` bindings for `c$9` and `c$7` while simultaneously generating an association list mapping `c$9` to `c$5` and `c$7` also to `c$5`. Thus, our program is now:

```
(letrec ([a$10 (lambda () (if (= rax 40) (c$7) (a$8)))]
         [a$8 (lambda () (if (= rax 32) (c$5) (a$6)))]
         [c$5 (lambda () (begin (set! rax (+ rax rax)) (r15)))]
         [a$6 (lambda () (begin (set! rax (* rax 2)) (r15)))])
  (begin (set! rax 40) (if (= rax 48) (c$5) (a$10))))
```

and the association list is:

```
((c$9 . c$5) (c$7 . c$5))
```

Next we recur through the remaining bodies, rewriting jump targets as directed by the association list, yielding the final result below.

```
(letrec ([a$10 (lambda () (if (= rax 40) (c$5) (a$8)))]
         [a$8 (lambda () (if (= rax 32) (c$5) (a$6)))]
         [c$5 (lambda () (begin (set! rax (+ rax rax)) (r15)))]
         [a$6 (lambda () (begin (set! rax (* rax 2)) (r15)))])
  (begin (set! rax 40) (if (= rax 48) (c$5) (a$10))))
```

While building the association list and removing bindings, the pass must be careful not to remove all of the jumps that participate in a cycle. For example, if the `expose-basic-blocks` output

```
(letrec ([f$1 (lambda () (f$2))]
         [f$2 (lambda () (f$1))])
  (f$1))
```

is passed to `optimize-jumps`, it must not eliminate both bindings, although it can remove one of the two, producing either

```
(letrec ([f$2 (lambda () (f$2))]) (f$2))
```

or the isomorphic

```
(letrec ([f$1 (lambda () (f$1))]) (f$1))
```

This pass should be added to your compiler just after `expose-basic-blocks`. The grammar for `optimize-jumps` is the same as the output grammar for `expose-basic-blocks`. In general, we should be able to enable or disable any optimization pass at will, so its input and output grammars must be the same.

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 tests11.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.

Use `make-begin` or `make-nopless-begin` to avoid nested `begin` expressions.