Assignment 7

Modified Tue Feb 24 12:35:21 EST 2009

Contents

1. Background

In this assignment we make one change to our source language: the addition of nontail calls. Though a simple change to the language, it impacts the compiler in many ways.

2. Calling Conventions

To handle nontail calls, we need to add a couple of additional notes to the calling conventions given in :

- at the point of a nontail call, the caller is responsible for positioning the frame pointer above its own frame to establish the callee's frame and placing the callee's frame arguments in the callee's frame; and
- at the point of return, the callee is responsible for ensuring that the frame pointer is again pointing at the base of its frame.

3. Scheme Subset 6

A grammar for the subset of Scheme we'll be handling this week adds nontail calls in *Value* and *Effect* contexts.

```
Program  ⟶  (letrec ([label (lambda (uvar*) Body)]*) Body)
Body     ⟶  (locals (uvar*) Tail)
Tail     ⟶  Triv
          |  (binop Value Value)
          |  (Value Value*)
          |  (if Pred Tail Tail)
          |  (begin Effect* Tail)
Pred     ⟶  (true)
          |  (false)
          |  (relop Value Value)
          |  (if Pred Pred Pred)
          |  (begin Effect* Pred)
Effect   ⟶  (nop)
          |  (set! uvar Value)
          |  (Value Value*)
          |  (if Pred Effect Effect)
          |  (begin Effect* Effect)
Value    ⟶  Triv
          |  (binop Value Value)
          |  (Value Value*)
          |  (if Pred Value Value)
          |  (begin Effect* Value)
Triv     ⟶  uvar | int | label
```

Unique variables (*uvar*), labels (*label*), integers (*int*), binary operators (*binop*), and relational operators (*relop*) are unchanged from the preceding subset. The machine constraints on integer values also remain from the preceding subset.

4. Things to do

To handle the new source language, we need to make minor updates to:

- `verify-scheme`,
- `remove-complex-opera*`,
- `flatten-set!`,
- `select-instructions`,
- `uncover-register-conflict`,
- `assign-registers`,
- `assign-frame`,
- `finalize-frame-locations`, and
- `finalize-locations`;

make major updates to:

- `impose-calling-conventions`,
- `uncover-frame-conflict`, and

- `discard-call-live`;

rewrite:

- `expose-frame-var`;

add:

- `pre-assign-frame`, and
- `assign-new-frame`;

discard:

- `introduce-allocation-forms`.

and make a small change in the iteration of our register and frame allocation passes.

This looks like a lot of work, and it is. Fortunately, the only totally new pass is `assign-new-frame`, since `pre-assign-frame` is essentially the same as `assign-frame`.

The new and updated passes are described in sections 4.1 through 4.15. The change in iteration is described in Section 4.16.

4.1. `verify-scheme`

This pass requires two simple one `match`-clause additions to handle nontail calls in *Value* and *Effect* contexts.

4.2. `remove-complex-opera*`

This pass requires two simple one `match`-clause additions to handle nontail calls in *Value* and *Effect* contexts.

4.3. `flatten-set!`

This pass requires two simple one `match`-clause additions to handle nontail calls in *Value* and *Effect* contexts.

A grammar for the output of this pass is given below.

```
Program  ⟶  (letrec ([label (lambda (uvar*) Body)]*) Body)
Body     ⟶  (locals (uvar*) Tail)
Tail     ⟶  Triv
          |  (binop Triv Triv)
          |  (Triv Triv*)
          |  (if Pred Tail Tail)
          |  (begin Effect* Tail)
Pred     ⟶  (true)
          |  (false)
          |  (relop Triv Triv)
          |  (if Pred Pred Pred)
          |  (begin Effect* Pred)
Effect   ⟶  (nop)
          |  (set! uvar Triv)
          |  (set! uvar (binop Triv Triv))
          |  (set! uvar (Triv Triv*))
          |  (Triv Triv*)
          |  (if Pred Effect Effect)
          |  (begin Effect* Effect)
Triv     ⟶  uvar | int | label
```

4.4. `impose-calling-conventions`

This pass also requires two one `match`-clause additions to handle nontail calls, plus it must record the lists of outgoing frame variables for the nontail calls it finds. (If your version of this pass does not already process *Pred* and *Effect* expression, it will now have to do so.)

For a nontail call in *Effect* context, this pass should perform the following transformation:

```
(proc e_0 ... e_{n-1} e_n ... e_{n+k-1})
  ⇒  (return-point rp-label
        (begin
          (set! nfv_0 e_n)
          ...
          (set! nfv_{k-1} e_{n+k-1})
          (set! p_0 e_0)
          ...
          (set! p_{n-1} e_{n-1})
          (set! ra rp-label)
          (proc fp ra p_0 ... p_{n-1} nfv_0 ... fv_{k-1})))
```

where the variables nfv_0 through nfv_{k-1} are fresh unique variables and *rp-label* is a fresh label.

The list of variables (nfv_0 ... nfv_{k-1}) must also be recorded as one of the new frames in the `new-frames` form (see grammar below) wrapped around the body. This is most easily accomplished by adding them to a running list stored in a variable that is local to the `Body` helper. All of the other helpers will need to be tucked inside `Body` as well.

We use new-frame variables rather than frame variables because we do not yet know where the outgoing frame will be situated, i.e., how big the caller's frame will be. This will be decided by the new pass `assign-new-frame`.

The relative order of the assignments for the new-frame variables is irrelevant, as is the order of the assignments for the register variables. All of the new-frame assignments should come before all of the register assignments, however, to limit the live ranges of the the parameter registers.

For a nontail call on the right-hand side of an assignment, this pass should perform essentially the same transformation as above, except follow the `return-point` form with an assignment of the left-hand-side variable to the return-value register, which is where the callee will leave its value.

```
(set! uvar
  (proc e0 ... en-1 en ... en+k-1))
  ⇒  (begin
        return-point code, as above
        (set! uvar rv))
```

The easist way to do this is to call back into the `Effect` helper to handle the call, then build the `begin` expression with the assignment, using something like the following `match` clause.

```
  [(set! ,var (,rator ,rand* ...))
   (make-begin
     `(,(Effect `(,rator ,rand* ...))
       (set! ,var ,return-value-register)))]
```

The new `new-frames` form wrapped around the body lists the new-frame variables (in order) for each of the frames found in the body. For example:

```
(new-frames ((nfv.17 nfv.18 nfv.19) (nfv.20)) tail)
```

declares that *tail* contains two `return-point` forms, one of which has three outgoing new-frame parameters (*nfv.17*, *nfv.18*, and *nfv.19*) and the other of which has one new-frame parameter (*nfv.20*). The lists need not appear in any particular order, but within each list, the first outgoing parameter must be listed first, the second next, and so on.

One more thing must be done by this pass, which is to include each of the new-frame variables in the `locals` form. If this is not done, `uncover-frame-conflict` will not know to create associations for them in the conflict table.

A grammar for the output of this pass is given below.

```
Program  ⟶  (letrec ([label (lambda () Body)]*) Body)
Body     ⟶  (locals (uvar*)
               (new-frames (Frame*) Tail))
Frame    ⟶  (uvar*)
Tail     ⟶  (Triv Loc*)
          |  (if Pred Tail Tail)
          |  (begin Effect* Tail)
Pred     ⟶  (true)
          |  (false)
          |  (relop Triv Triv)
          |  (if Pred Pred Pred)
          |  (begin Effect* Pred)
Effect   ⟶  (nop)
          |  (set! Var Triv)
          |  (set! Var (binop Triv Triv))
          |  (return-point label (Triv Loc*))
          |  (if Pred Effect Effect)
          |  (begin Effect* Effect)
Loc      ⟶  reg | fvar
Var      ⟶  uvar | Loc
Triv     ⟶  Var | int | label
```

4.5. `uncover-frame-conflict`

This pass requires a single `match`-clause addition to handle nontail calls, i.e., `return-point` forms. In addition, it must determine the set of *call-live* variables and frame locations. A variable or frame location is call-live if it is live across a call, i.e., its value might yet be needed at the return point of the call.

The set of `call-live` variables and frame locations is recorded in a new `call-live` body wrapper for use by `assign-new-frame`. The set of call-live variables (but not frame locations) in recorded separately in a `spills` wrapper for use by `pre-assign-frame`.

In practical terms, call-live variables and frame locations are those that are live after some `return-point` form. Gathering the entire set of call-live variables is most easily accomplished by having the `Effect` helper that processes `return-point` forms add them to a running set stored in a variable that is local to the `Body` helper. All of the other helpers will need to be tucked inside `Body` as well.

The set of variables and frame locations live on entry to a `return-point` form includes both those that are live on exit from the `return-point` form (the call-live variables and frame locations) and those that are live one entry to the *Tail* expression tucked inside the `return-point` form, which should be processed just like any other *Tail* expression.

Only the *Body* grammar is different in the output of this pass. It differs in the addition of the `spills`, `call-live`, and (as before) `frame-conflict` wrappers.

```
Body  ⟶  (locals (uvar*)
            (new-frames (Frame*)
              (spills (uvar*)
                (frame-conflict conflict-graph
                  (call-live (uvar/fvar*) Tail)))))
```

4.6. `pre-assign-frame`

This pass finds frame homes for the variables listed in the `spills` list. It differs from `assign-frame` only in the structure of the input and output *Body* forms. Like `assign-frame`, it eliminates the `spills` form but leaves behind the other forms. It also adds the `locate` form, which is not present in its input. So the *Body* grammar for the output is as shown below.

```
Body  ⟶  (locals (uvar*)
            (new-frames (Frame*)
              (locate ([uvar fvar]*)
                (frame-conflict conflict-graph
                  (call-live (uvar/fvar*) Tail)))))
```

4.7. `assign-new-frame`

For each body, this pass determines the size the body's frame, based on the locations of the variables and frame variables in the `call-live` list. It then selects homes for each of the new-frame variables listed in the `new-frames` form and rewrites the code of the body to place the callee's frame just above the body's frame.

The size of the frame is, simply, one more than the maximum index of the frame locations of the call-live variables or frame variables. The index of a frame variable can be deteremined directly via the helpers.ss procedure *frame-var-¿index*, while determining the index of a variable involves first finding, via the `locate` form, the frame variable to which it has been assigned.

Once the frame size n has been determined, each outgoing sequence of new-frame variables is assigned to the frame locations fv_n, fv_{n+1}, etc. These are recorded in the `locate` form along with the assignments already made there by the preceding pass. In addition, each `return-point` form is rewritten to increment and decrement the frame pointer register by the number of bytes nb represented by the n items in the procedure's frame.

```
(return-point rp-label tail)  ⇒
  (begin
    (set! fp (+ fp nb))
    (return-point rp-label tail)
    (set! fp (- fp nb)))
```

nb may be computed by shifting *n* left by the value of the helpers.ss variable `align-shift`.

For example, consider the following *Body*.

```
(locals (nfv.17 nfv.18 nfv.19 nfv.20 local ...)
  (new-frames ((nfv.17 nfv.18 nfv.19) (nfv.20))
    (locate ([rp.12 fv0] [x.3 fv2])
      (frame-confict conflict-table
        (call-live (rp.12 x.3 fv1)
          tail)))))
```

The frame size is 3, because the maximum index of the locations of the call-live variables and frame variables is 2. (The index for `rp.12` is 0, for `x.3` is 2, and for `fv1` is 1. The maximum of 0, 2, and 1 is 2.) The locations assigned each sequence of frame variables therefore starts with `fv3`, as shown in the output below.

```
(locals (local ...)
  (ulocals ()
    (locate ([rp.12 fv0] [x.3 fv2]
             [hfv.17 fv3] [nfv.18 fv4] [nfv.19 fv5]
             [nfv.20 fv3])
      (frame-confict conflict-table
        tail))))
```

The `new-frames` and `call-live` forms, which are no longer needed, have been dropped. Also, the new-frame variables have been removed from the `locals` form, which is a simple matter of subtracting all of the new-frame variables listed in the `new-frames` form from the set of incoming locals.

One additional form has appeared: the `ulocals` form formerly inserted by `introduce-allocation-forms`, which we can now discard from our compiler.

Within *tail* in the example above, the amount by which the frame pointer is incremented and decremented around each `return-point` form is 3 scaled by `align-shift`, or 24 given the standard `helpers.ss` value of 3 for `align-shift`.

Ideally, we would determine the frame sizes independently at each nontail call site, rather than taking a "one size fits all" approach that may result in overly large frames at some call sites. This would require `uncover-frame-conflict` to leave behind more detailed `call-live` information, and it would make `assign-new-frame` significantly more complex as well. We've decided to hold the complexity down instead.

4.8. `introduce-allocation-forms`

This pass goes away, i.e., no longer appears in the set of passes declared via the `compiler-passes` parameter. The `locate` form it formerly introduced is now introduced by `pre-assign-frame`, and the `ulocals` form is now introduced by `assign-new-frame`.

4.9. `select-instructions`

This pass requires a simple one `match`-clause addition to handle `return-point` forms in *Effect* context.

4.10. `uncover-register-conflict`

This pass requires a simple one `match`-clause addition to handle `return-point` forms in *Effect* context.

A nontail call is an implicit assignment to all of the caller-save registers. In the case of `uncover-frame-conflict`, we do not care about conflicts involving registers. For `uncover-register-conflict`, only registers can be live after a call (since all call-live variables have been spilled), and we do not care about register-register conflicts. Thus we do not have to worry about the fact that a nontail call is an implicit assignment, although we could verify that no registers, other than the frame-pointer and return-value registers, are live after a `return-point` form.

4.11. `finalize-frame-locations`

This pass requires a simple one `match`-clause addition to handle `return-point` forms in *Effect* context. It also needs to process the set of live locations listed in each call, as if they were arbitrary *Triv* expressions, in case any new-frame variables are present in the set of live locations. For example,

(*proc rbp r15 r8 r9 nfv.20*)

should be converted to

(*proc rbp r15 r8 r9 fv3*)

if `nfv.20` has been assigned to frame variable `fv3`.

If this were not done, `uncover-register-conflict` would trip over *nfv.20* when it appeared in its live set.

4.12. `discard-call-live`

This pass must now process *Effect* and *Pred* expressions to look for nontail calls, whereas previously all calls were in *Tail* context.

4.13. `finalize-locations`

This pass requires a simple one `match`-clause addition to handle `return-point` forms in *Effect* context.

4.14. `expose-frame-var`

This pass must now be flow sensitive to track changes in frame pointer register. This is necessary because we are now incrementing the frame pointer before each `return-point` form, yet frame variables may appear within the `return-point` form. The index of a frame variable appearing within code affected by one of these frame-pointer increments must be adjusted by the amount of the increment. To make this fully general, the pass should be able to handle arbitrary placement of the frame-pointer increments and decrements, in case an optimization pass has moved or altered them in some way, e.g., to leave the frame-pointer in place if two or more nontail calls occur in succession.

Each helper used by this pass will need to be told the amount by which the frame pointer is currently offset. It will also need to return the final offset as an additional return value, i.e., return both the transformed expression and the final frame-pointer offset. The `Effect` helper will need to recognize frame-pointer increments and decrements and adjust the offset accordingly. For *if* expressions, the `then` and *else* parts should start off with the same offset, i.e., the offset coming out of the *test* part, and it's a good idea to verify that the ending offsets for the *then* and *else* parts are the same.

4.15. `expose-basic-blocks`

This pass requires a one `match`-clause addition to handle `(return-point` *rp-label tail*`)` forms in *Effect* context.

It should package up the "rest" expressions (those that follow the `return-point` form) in a `lambda` expression and bind *rp-label* to this `lambda` expression, process the encapsulated *tail* expression like any other *Tail* expression, and process the "before" expressions (those that precede the `return-point` form, if any, in the same `begin` expression) as with any other *Effect* expression.

4.16. Iterating

Because some variables (the call-live variables) are given frame homes before the iteration begins, we need to make a simple change to the iteration, which is to move `finalize-frame-locations` to the top of the iteration, where it is run each time, from the bottom, where it was run only after `assign-frame`. It needs to be run at the start of the iteration to finalize the locations assigned by `pre-assign-frame` and `assign-new-frame`. With this change and the addition of the new passes, the `compiler-passes` parameter is set as follows.

```
(compiler-passes '(
  verify-scheme
  remove-complex-opera*
  flatten-set!
  impose-calling-conventions
  uncover-frame-conflict
  pre-assign-frame
  assign-new-frame
  (iterate
    finalize-frame-locations
    select-instructions
    uncover-register-conflict
    assign-registers
    (break when everybody-home?)
    assign-frame)
  discard-call-live
  finalize-locations
  expose-frame-var
  expose-basic-blocks
  flatten-program
  generate-x86-64
))
```

5. Boilerplate and Run-time Code

The boilerplate and run-time code does not change.

6. Testing

A small set of invalid and valid tests for this assignment have been posted in tests7.ss. You should make sure that your compiler passes work at least on this set of tests.

7. Coding Hints

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