Assignment 5

Modified Thu Feb 12 11:07:14 EST 2009

Contents

1. Background

The goal of the register and frame allocator is to place local variables into registers when possible and into frame locations otherwise. Variables placed into frame locations are called *spilled variables* or, simply, *spills* because they do not fit into the register set. Spilled variables are no longer candidates for registers, but whenever a spilled variable is placed into a frame location, additional local variables may need to be added to serve as load and store temporaries for the spilled variables. For example, on the x86_64, if both x and y are spilled in

```
(set! x (+ x y))
```

a temporary for x or one for y must be introduced, i.e.,

```
(set! u y)
(set! x (+ x u))
```

Such temporaries are called *unspillable* variables, because they must be assigned to registers. No more unspillables are alive at a time than the number of available registers, so it is always possible to do so. Variables that can be spilled if necessary are referred to as *spillable* variables.

At the end of register allocation, some of the original local variables and all of the unspillable variables are in register homes, while the remainder of the original local variables are in frame homes.

The main part of the register and frame allocator is an iterative process that repeatedly attempts to assign as many variables to registers as possible and spills those that don't fit.

A couple of preparatory tasks occur prior to the iterative process.

1. The program is rewritten to reflect the calling conventions of the target architecture and run-time system, e.g., register and frame parameters are made explicit. During this process, the locations of outgoing nontail-call frame parameters are left unspecified, since the frame size at a nontail call cannot be determined yet.

1. A frame conflict graph is built, associating each local with a list of the other locals and frame locations with which it conflicts. During this phase, all locals are pessimistically asssumed to be destined for the frame, so these sets relate local variables with each other and with the fixed frame locations. Building conflict sets involves live analysis.

3. Variables live across a nontail call (referred to as "call-live" variables) are identified as spills, and each is assigned a specific frame location.

4. The location of each block of nontail-call frame arguments is determined, based on the locations of the call-live variables. Each block is placed above the variables that are live across the corresponding call. The amount by which the frame pointer must be incremented and decremented around the call is determined at this point as well.

The iterative process itself consists of four tasks.

5. The instructions to be used for each operation are identified, and unspillable variables are introduced where necessary, e.g., to replace memory operands with registers where registers are required. Spillable as well as unspillable variables are optimistically assumed to be registers.

6. A register conflict graph is built, associating each variable with a list of the variables and registers with which it conflicts.

7. An attempt is made to assign each unspillable and spillable variable to a register. If this is successful, the iteration terminates and the compiler moves on to the passes after register/frame allocation. Otherwise, some of the spillable variables are spilled and the iteration continues with frame allocation.

8. Each spill is assigned to a frame location, using the frame conflict and move sets created prior to the start of the iteration. The process continues with Step 5.

Our new passes perform tasks 2, 5 and 8. We handled tasks 6 and 7 in the preceding assignment. We are effectively doing tasks 1, 3 and 4 manually as we write our test cases, but we'll modify the compiler to take these tasks over in subsequent assignments.

2. Scheme Subset 5

The grammar for the subset of Scheme we'll be handling this week is the same as the grammar for last week:

```
Program  ⟶  (letrec ([label (lambda () Body)]*) Body)
Body     ⟶  (locals (uvar*) Tail)
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))
          |  (if Pred Effect Effect)
          |  (begin Effect* Effect)
Loc      ⟶  reg | fvar
Var      ⟶  uvar | Loc
Triv     ⟶  Var | int | label
```

Unique variables (*uvar*), registers (*reg*), frame variables (*fvar*), labels (*label*), integers (*int*), binary operators (*binop*), and relational operators (*relop*) are unchanged from the preceding subset.

The only difference is that, with this subset, the grammar expressions are no longer limited by all of the constraints of the x86_64 target architecture, since our new instruction selection pass will rewrite the code as necessary so that it obeys most of those constraints.

Two constraint do remain, however: the second operand of `sra` must still be an exact integer constant k, $0 \leq k \leq 63$, and every other integer constant must be an exact integer n, $-2^{63} \leq n \leq 2^{63} - 1$.

3. Things to do

To handle the new source language, we need to update the verifier (to remove the checks for most target-machine constraints), make superficial modifications to:

```
  uncover-register-conflict
```

make minor modifications to:

```
  assign-registers
  finalize-locations
```

and add several new passes:

```
  uncover-frame-conflict
  introduce-allocation-forms
  select-instructions
  assign-frame
  finalize-frame-locations
```

This seems like a lot do do, but fortunately all but two of the new passes are adaptations of existing passes. The two entirely new passes are `introduce-allocation-forms` and `select-instructions`. The first of these is trivial, while the second is challenging. The changes to the existing passes are straightforward.

Section 3.1 describes how these passes work together to perform the iterative register and frame allocation process, and the new and modifed passes are described in Sections 3.2-3.10.

3.1. Iterating

If the register allocator fails to assign all variables to registers, it spills some to the frame. These are given frame homes, unspillable variables are introduced if necessary to load and store the spills, and the register allocator is rerun. This process is repeated until register allocation succeeds. While this typically requires one or three iterations, the actual number of iterations varies.

Until now, we have been able to use a flat list of passes to describe to the driver the structure of the compiler. Now, however, we must express iteration of the register- and frame-allocation passes. To do so, we use the full generality of the `compiler-passes` parameter, which is called as shown below.

`(compiler-passes '(`*spec* `...))`

Each *spec* must take one of the following forms.

- *pass-name*: This tells the driver to run the named pass on the output of the preceding pass to produce a new version of the intermediate code. This is the only form we've used to date.
- `(iterate` *spec* `...)`: This tells the driver to run *spec* `...` repeatedly, using the output of the last *spec* as input to the first *spec* each time it reruns *spec* `...`.

- `(break when` *predicate*`)`: This tells the driver to break out of the current iteration (or out of the compiler if not within an `iterate` form) if *predicate* returns true when applied to the current intermediate program.

For example,

```
(compiler-passes '(a (iterate b (break when c?) d) e))
```

causes the driver to run `a` on the input and feed its output to `b`. If `c?` applied to the output of `b` is true, the driver feeds the output of `b` to `e`; otherwise, the driver feeds the output of `b` to `d` and the output of `d` to `b`, then tests `c?` on the output of `b` again, and so on.

In our compiler, we need to iterate until each `lambda` expression is complete, i.e., all of its variables have been assigned register or frame locations. It may be that the variables of some `lambda` expressions in a program will have homes after one pass, others after two passes, and others not until more than two passes. To see if all of the `lambda` expressions are complete, we can use the following predicate, `everybody-home?`.

```
(define-who everybody-home?
  (define all-home?
    (lambda (body)
      (match body
        [(locals (,local* ...)
           (ulocals (,ulocal* ...)
             (spills (,spill* ...)
               (locate (,home* ...)
                 (frame-conflict ,ct ,tail))))) #f]
        [(locate (,home* ...) ,tail) #t]
        [,x (error who "invalid Body ~s" x)])))
  (lambda (x)
    (match x
       [(letrec ([,label* (lambda () ,body*)] ...) ,body)
        (andmap all-home? `(,body ,body* ...))]
       [,x (error who "invalid Program ~s" x)])))
```

When `assign-registers` succeeds in assigning registers to all (remaining) variables, it records the register homes in the `locate` form, along with any frame homes already recorded there. Because they are no longer needed, it also drops the `locals`, `ulocals`, `frame-conflict`, and `register-conflict` forms, leaving just the `locate` form wrapped around the body.

Thus, the `everybody-home?` predicate simply looks to see if the these other forms are present in any `lambda` body or the `letrec` body, and if not declares that everybody is indeed home.

Once we have this predicate, we can define our compiler passes for the driver as follows.

```
(compiler-passes '(
  verify-scheme
  uncover-frame-conflict
  introduce-allocation-forms
  (iterate
    select-instructions
    uncover-register-conflict
    assign-registers
    (break when everybody-home?)
    assign-frame
    finalize-frame-locations)
  discard-call-live
  finalize-locations
  expose-frame-var
  expose-basic-blocks
  flatten-program
  generate-x86-64
))
```

3.2. verify-scheme

This pass must be modified to eliminate or adjust checks to verify that the input satisifies the machine constraints.

3.3. uncover-frame-conflict

This pass records for each variable the set of other variables and frame locations (frame variables) with which it conflicts. This pass and `uncover-register-conflict` are similar. The only difference with respect to the handling of body expressions is that where `uncover-register-conflict` looks for registers with `register?`, `uncover-frame-conflict` looks for frame variables using `frame-var?`. You may wish to abstract the conflict uncovering code away from this predicate to create a helper that both can use.

Frame conflicts are recorded in a new `frame-conflict` form wrapped around the body. Thus, the only change in the grammar is in the *Body* nonterminal, which now reads as follows.

```
Body  ⟶  (locals (uvar*)
            (frame-conflict conflict-graph Tail))
```

3.4. introduce-allocation-forms

On each iteration of register and frame allocation, the instruction selection pass adds unspillable variables to the set of variables needing register homes, and the frame allocator assigns locations to spilled variables. The former are recorded in a `ulocal` form, while the latter are recorded in a `locate` form.

Since these forms are present on the second and subsequent iterations, it's convenient for them to be there on the first iteration as well. So `introduce-allocation-forms` simply wraps each body with a `ulocal` form with an empty list of unspillables and a `locate` form with an empty list of bindings. This affects only the *Body* nonterminal of the grammar as shown below.

```
Body  ⟶  (locals (uvar*)
            (ulocals ()
              (locate ()
                (frame-conflict conflict-graph Tail))))
```

This is a trivial pass that operates superficially on each *Body* but does not venture into the *Tail* expression within the *Body*.

3.5. select-instructions

Computer architectures place restrictions on the sizes of constant operands and the contexts in constant or memory operands can appear, in addition to occasional restrictions on which registers can be used in particular contexts. These restrictions should never impact what can be expressed in the high-level source language, so at some point the compiler must rewrite the incoming code so that it obeys the restrictions of the target architecture. In some cases, the rewriting process requires the compiler to introduce temporaries that must be assigned to registers, so the logical place for this process to occur is prior to register allocation.

While the actual instructions are not generated until the code generator runs, rewriting the code to obey the target-architecture restrictions virtually selects instructions, so this process is typically referred to as *instruction selection.*

This pass should assume that each unique variable *uvar* will get a register. If this turns out not to be the case, the pass will be run again after the variables spilled by the register allocator have been replaced with fixed frame variables.

The output-language grammar for this pass is identical to the input-language grammar, but all of the architecture restrictions that we dropped in our source language from the preceding assignment are back in force after this pass.

There may be no elegant way to code this pass because it necessarily reflects the quirks of the x86_64 architecture. We suggest that you have each of the helpers return two values: the rewritten expression and the set of unspillables inserted into the rewritten code. Create new unspillable variables with the helpers.ss procedure `unique-name`. In our implementation, we use `(unique-name 't)` so that each unspillable looks like `t.`*n*, but the actual names are not important as long as the are *uvar*s. Also, use the helpers.ss procedure `make-begin` wherever your pass might need to create a `begin` expression so that you don't end up with unnecessary or nested `begin` expressions.

You might want to avoid the introduction of unnecessary unspillables when possible by taking into account the commutativity of the operators `+`, `*`, `logand`, and `logor`. For example, if faced with `(set! x.5 (- 3 x.5))` you must convert this to something like:

```
(begin
  (set! t.6 3)
  (set! t.6 (- t.6 x.5))
  (set! x.5 t.6))
```

where `t.6` is a new unspillable. For `(set! x.5 (+ 3 x.5))`, however, you can simply swap the operands:

```
(set! x (+ x 3))
```

Similarly, you can avoid unnecessary unspillables for some relational operator calls by swapping the operands and inverting the sense of the comparison, so `(< 3 x.5)` becomes `(> x.5 3)`.

Last week's `verify-scheme` pass might be a good resource for some of the tests `select-instructions` must make.

The input-language grammar for this pass is slightly different from the output-language grammar for `introduce-allocation-forms` because the input for this may also come from a preceding iteration of the register allocator. When this occurs, register and frame allocation has not been completed for one or more of the bodies, but it may have been completed for others, since the number of iterations required for any two bodies may be different. Furthermore, the list of unspillables in the `ulocals` form and the list of bindings in the `locate` form may be nonempty. This is reflected in the syntax for *Body*.

```
Body  ⟶  (locals (uvar*)
            (ulocals (uvar*)
              (locate ([uvar fvar]*)
                (frame-conflict conflict-graph Tail))))
      |  (locate ([uvar Loc]*) Tail)
```

The first, *incomplete*, form of *Body* is for bodies still requiring more work by the register and frame allocator; the second, *complete*, form is for bodies whose variables have all been given locations. For the latter, this pass should simply reproduce the body form in the output.

The output-language grammar for this pass is the same as the input-language grammar. As noted above, however, the actual syntax of a program output from this pass reflects the x86_64 architecture constraints.

3.6. `uncover-register-conflict`

As in the preceding assignment, the goal of this pass is to record conflicts between each variable and the variables and registers with which it conflicts. This version differs only superficially, in that there are now more wrappers on *Body* expressions, as reflected in the grammar for *Body* given above.

For a completed body, i.e., one with just a `locate` wrapper, this pass should simply reproduce the body form in the output. For an incomplete body, it should process *Tail* expressions and insert the `register-conflict` form as in the preceding version. The output-language grammar differs from the input-language grammar only in the appearance of the `register-conflict` form in the syntax of an incomplete *Body*.

```
Body  ⟶  (locals (uvar*)
           (ulocals (uvar*)
             (locate ([uvar fvar]*)
               (frame-conflict conflict-graph
                 (register-conflict conflict-graph Tail)))))
      |  (locate ([uvar Loc]*) Tail)
```

3.7. `assign-registers`

This pass changes in three ways from the last assignment.

- The *Body* syntax changes in essentially the same way as for `uncover-register-conflict`,
- The register-assignment algorithm much take into account the unspillable variables listed in the `ulocals` form.
- When variable are spilled, it should produce a list of spills in the output, using a new `spills` wrapper, rather than aborting with an error.

The input grammar is the same as the `uncover-register-conflict` output grammar. Again, this pass should reproduce in the output any complete bodies.

For incomplete bodies, what it produces depends on whether it was able to find registers for all of the spillable and unspillable variables listed in the `locals` and `ulocals` forms. If so, it should produce a complete body with just the `locate` form listing both the original frame locations and the new register locations. Otherwise, it should produce an incomplete body with a list of the spilled variables in a `spills` form. It *must not* record the assignments it was able to make, since the assignments will likely have to change on the next iteration. It *must*, however, remove the spilled variable from the `locals` form, since they are no longer candidates for registers. The set of spills and the set of spillables should be disjoint and their union should be the same as the original set of spillables. The unspillables list should not change. This pass must also drop the `register-conflict` form, since this will certainly change on the next iteration. The output-language grammar is thus as follows.

```
Body  ⟶  (locals (uvar*)
           (ulocals (uvar*)
             (spills (uvar*)
               (locate ([uvar fvar]*)
                 (frame-conflict conflict-graph Tail)))))
      |  (locate ([uvar Loc]*) Tail)
```

This pass must include the unspillable variables listed in the `ulocals` form in the set of variables for which it tries to find register homes. It must also avoid choosing an unspillable variable when forced to remove a high-degree node from the graph, since unspillable variables *must* get registers. This is guaranteed to work as long as even a few registers are provided, since unspillable variables have short live ranges and thus conflict with few other unspillable variables.

3.8. `assign-frame`

This pass is called if `assign-registers` was unsuccessful in finding register homes for all of the local variables. The set to which frame locations must be assigned is identified in the `spills` wrapper produced by `assign-registers`.

Frame allocation is similar to register allocation, but simpler in that all nodes are low-degree, since the number of frame locations is effectively infinite. This means that no equivalent of spilling occurs. It also means that we can simplify all of the nodes from the graph immediately, i.e., we can skip simplification and go straight to the selection.

On the other hand, the simplify phase of the register allocator can be viewed as a way of sorting the variables into an order intended to increase the number of variables assigned to registers, i.e., decrease the number of spills, versus a random order. Sorting the variables on input to the frame allocator may be worthwhile as well, and you may wish to experiment with different strategies to see what seems to work best. For example, if we were recording moves between variables, we would want to select locations first for spills that are move related, directly or indirectly, to fixed frame locations. This reduces the possibility that a conflicting variable with no particular need for that frame location will be assigned to that location instead.

The select phase assigns the first variable in the list of spills to a compatible frame location (frame variable). A variable x is compatible with a frame variable *fv* if x does not directly conflict with *fv* (in the frame conflict set) and if x does not conflict with a variable y previously assigned to *fv*. When making the latter determination, the homes assigned to frame variables on previous iterations, which are recorded in the incoming `locate` form, must be taken into account.

An easy way to find a compatible location is to determine the set of frame variables to which x cannot be assigned, then loop through the frame variables starting with `fv0` to find one that is not in that set. The helpers.ss procedure `index->frame-var` should simplify this task.

The assignments made by this pass should be added to those already present in the `locate` form. The output grammar is the same as for the output grammar of `assign-registers` except that the `spills` form is dropped.

```
Body  ⟶  (locals (uvar*)
            (ulocals (uvar*)
              (locate ([uvar fvar]*)
                (frame-conflict conflict-graph Tail))))
      |  (locate ([uvar Loc]*) Tail)
```

3.9. `finalize-frame-locations`

This pass is called after `assign-frame` to replace each occurrence of a frame-allocated variable with the corresponding frame variable, based on the `locate` form. It is essentially the same as `finalize-locations`, except that the *Body* forms again come in complete and incomplete forms, while the *Body* forms for the original are always in the complete form. Furthermore, this pass does not discard the `locate` form because the locations assigned to frame variables on this iteration of the register and frame allocation passes need to be taken into account by the next, if more variables are spilled. Thus, the output-language grammar for this pass is the same as the output-language grammar for the preceding pass.

One additional change must be made to this pass, which is that it must recognize when it would produce assignments of the form:

```
(set! fvi fvi)
```

When it would produce such an assignment, it should produce `(nop)` instead, since the assignment serves no purpose. These nops will be washed away later by `expose-basic-blocks`.

3.10. `finalize-locations`

The only change to this pass is to avoid producing a move from a location to itself, i.e., an assignment of the form:

(`set!` *loc*$_1$ *loc*$_2$)

where *loc*$_1$ and *loc*$_2$ are the same location, after any substitutions have been performed. So, for example,

```
(begin
  (set! x 15)
  (set! x (+ x 5))
  (set! x rax)
  (r15))
```

becomes:

```
(begin
  (set! rax 15)
  (set! rax (+ rax 5))
  (nop)
  (r15))
```

if `x` has been assigned to `rax`.

Even though the `locate` form may list frame locations for some variables, all of the frame substitutions should have been made during the iterative register and frame allocation process by `finalize-frame-locations`, so the only substitutions that should actually take place during this pass are register substitutions. This need not affect how the code is written, although you can take advantage of this fact and create an environment that contains only register assignments.

4. Boilerplate and Run-time Code

The boilerplate and run-time code does not change.

5. Testing

A small set of invalid and valid tests for this assignment have been posted in tests5.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, including ones containing multiple bodies, some of which require more than one iteration of the register and frame allocation passes. Here is one such example:

```
(letrec ([f$1 (lambda ()
                (locals (x.1 y.2 z.3)
                  (begin
                    (set! x.1 1)
                    (set! y.2 2)
                    (set! rax (+ x.1 y.2))
```

```
                   (r15 rax rcx rdx rbx rbp rdi rsi r8 r9 r10
                        r11 r12 r13 r14))))])
  (locals () (f$1 rbp r15)))
```

Rather than tackle all of the modifications at once, we suggest you make the modifications required for the existing passes; add `introduce-allocation-forms`; and throw in stub versions of `uncover-frame-conflict`, `select-instructions`, `assign-frame`, and `finalize-frame-locations`. Then test with programs that don't require the new passes to do anything, i.e., programs that don't violate the architecture constraints and don't spill any variables. Any valid tests from the preceding assignment should do.

Once you have this working, you can replace your stub versions one at a time with working versions. You may wish to leave `select-instructions` for last, since everything else is relatively straightforward, but be aware that a source program that does not appear to violate architectural constraints may do so after some variables have been spilled.