Assignment 6

Modified Thu Feb 19 12:25:42 EST 2009

Contents

1. Background

Now that the compiler is capable of finding registers and frame locations for our local variables, we can turn over to it the job of handling arguments and return values as well. This will relieve us from having to deal directly with registers and frame variables explicitly in our test cases. We'll also generalize our language to allow arbitrary nesting of expressions within procedure and primitive calls.

To handle arguments, the compiler rewrites the code so that the formal parameters are assigned to a specific set of register and frame locations and, at call points, arranges to assign this same set to the argument values. At return points, the compiler arranges to store the return value in a specific return-value register and to jump (tail call) the return address stored in a specific return-address register. The specific register and frame locations used for arguments, the return value, and the return address are determined by the *calling conventions*, which also determine the set of registers preserved by a callee (the "callee save" registers) and the set of registers that must be saved by the caller, if desired (the "caller-save" registers).

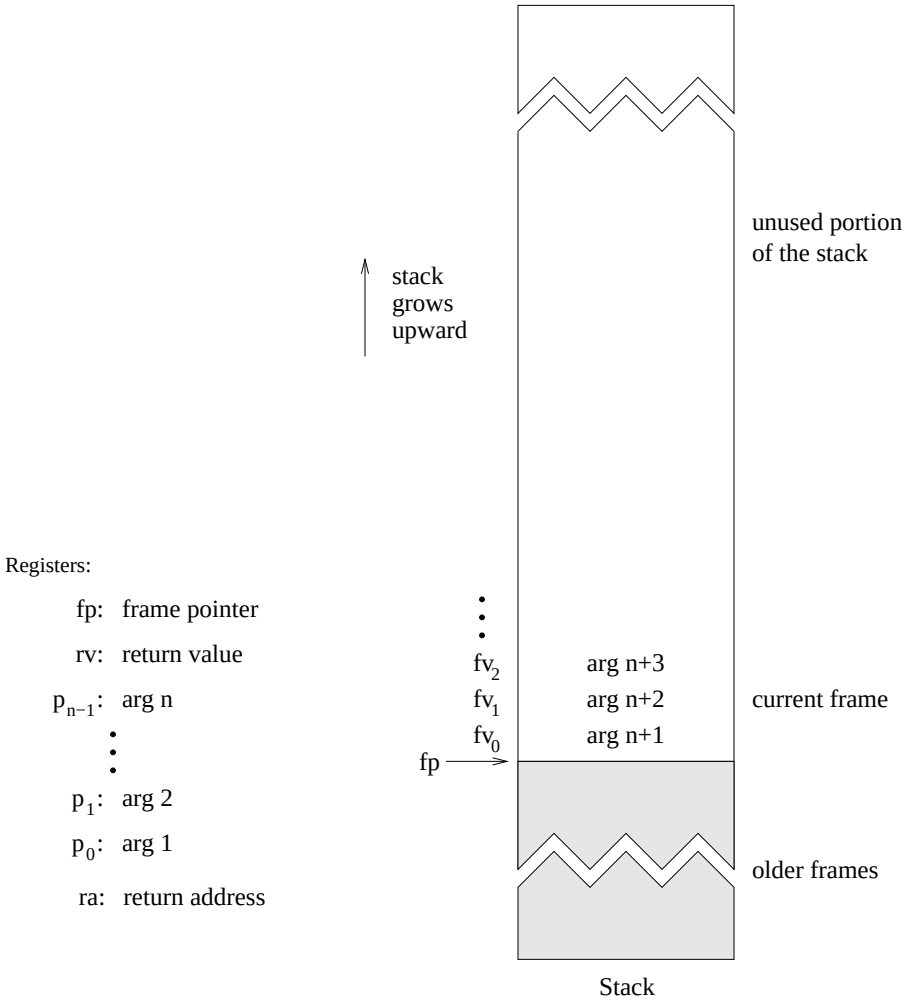
2. Calling Conventions

The standard x86_64 calling conventions are described in the System V Application Binary Interface AMD64 Architecture Processor Supplement, but since our procedures do not need to interact with procedures written in other languages, we will use our own, simpler conventions:

- the first n (possibly zero) arguments are passed in the registers given by the helpers.ss variable `parameter-registers`;
- the remaining arguments are passed in frame locations `fv0`, `fv1`, etc., which map to consecutive words based at the register given by the variable `frame-pointer-register`;
- the return address is passed in the register given by the variable `return-address-register`;
- the procedure's value is returned in the register given by the variable `return-value-register`; and
- all registers are caller-save.

The frame-pointer register might be considered callee-save because it must be pointing just above the caller's frame when the callee returns, just as it is pointing just above the caller's frame at the point of call. Yet, it is possible that the entire stack or portions of it are relocated during program execution, e.g., to support stack resizing or continuation capture and invocation. So the actual address in the frame-pointer register may be different even though its position relative to the base of the current frame is fixed. Thus, the frame pointer is more properly considered an implicit return value from the callee, and if the original location really needs to be preserved, the caller must do so.

The calling conventions are summarized by the following diagram.



3. Scheme Subset 6

The grammar for the subset of Scheme we'll be handling this week adds formal and actual parameters and does not permit explicit use of fixed register and frame locations. It also now allows the subexpressions of a procedure call, primitive call, or assignment to be arbitrarily nested *Value* expressions.

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

The subexpressions of a procedure or primitive call may be evaluated in any order, i.e., left-to-right, right-to-left, or any other order, as long as the evaluation of any two arguments is not interleaved. Interleaving is detectable only when effects are involved.

5. Things to do

To handle the new source language, we need to update `verify-scheme`; add three new passes:

- `remove-complex-opera*`,
- `flatten-set!`, and
- `impose-calling-conventions`;

and make a slight modification to `expose-frame-variable`.

5.1. `verify-scheme`

This pass must be modified to account for the grammar changes.

5.2. `remove-complex-opera*`

This pass removes nested primitive calls from within procedure calls and other primitive calls, making the argument values "trivial." Programs produced by this pass are in the language described by the grammar

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 Value*`)`
	|	`(if `*Pred Effect Effect*`)`
	|	`(begin `*Effect*`* `*Effect*`)`
Value	⟶	*Triv*
	|	`(`*binop Triv Triv*`)`
	|	`(if `*Pred Value Value*`)`
	|	`(begin `*Effect*`* `*Value*`)`
Triv	⟶	*uvar* | *int* | *label*

The only change from the preceding grammar is that the subexpressions of primitive calls and procedure calls are now *Triv* expressions rather than *Value* expressions.

In order to carry this out, each nontrivial *Value* must be assigned outside of the call to a fresh unique variable. For example:

```
(f$1 (+ (* x.2 x.5) 7) (sra x.1 3))
```

becomes

```
(begin
  (set! tmp.7 (* x.2 x.5))
  (set! tmp.6 (+ tmp.7 7))
  (set! tmp.8 (sra x.1 3))
  (f$1 tmp.6 tmp.8))
```

The set of new unique variables introduced during this process must be added to the `locals` list enclosing the body.

5.3. `flatten-set!`

This pass rewrites `set!` expressions as necessary to push them inside `if` and `begin` expressions so that, in the output, the right-hand-side of each `set!` contains neither `if` nor `begin` expressions. We do this to convert assignments into a form that more closely resembles assembly instructions. Programs produced by this pass should be in the language described by the following grammar, which differs only in that the right-hand side of a `set!` is (once again) restricted to a *Triv* or primitive call.

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*`))`
| `(if `*Pred Effect Effect*`)`
| `(begin `*Effect*`* `*Effect*`)`
Triv ⟶ *uvar* | *int* | *label*

When the right-hand side of a `set!` expression is a `begin` expression, the `set!` should be pushed inside of the `begin` with the last subexpression of the `begin` as its right-hand side.

`(set!` x `(begin` e_1 `...` e_{n-1} e_n`))` →
`(begin` e_1 `...` e_{n-1} `(set!` x e_n`))`

When the right-hand side is an `if` expression, the `set!` should be pushed inside the `if` and replicated so that both the consequent and alternative of the `if` become right-hand sides of the assignment.

`(set!` x `(if` e_1 e_2 e_3`))` →
`(if` e_1 `(set!` x e_2`) (set!` x e_3`))`

Of course, the new right-hand side in either case may be an `if` or `begin` expression, so it is necessary to recur until the right-hand side is something other than an `if` or `begin` expression.

5.4. `impose-calling-conventions`

This pass imposes the calling conventions on the output code. It arranges for each argument to be passed in a register or on the stack, as appropriate, and the return value to be returned in a register. After it is done, `lambda` expressions no longer have explicit formal parameters, and both calls and returns are reduced to the equivalent of jumps. The programs produced by this pass are in the same language as the input language for `uncover-frame-conflict`. This language is described by the following grammar.

```
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
```

Our particular choice of calling conventions should not be hard-wired into the code for this pass. Instead, the pass should assume only a fixed number (possibly zero) of parameter registers p_0, p_1, ..., p_{n-1}, a frame-pointer register *fp*, a return-value register *rv*, and a return-address register *ra*. The actual registers to use are given by the new helpers.ss variables `parameter-registers`, `frame-pointer-register`, `return-value-register`, and `return-address-register`.

This pass and the remainder of the compiler should be tested with different settings for these variables, including more, fewer, and different parameter registers, a different return-value register, a different frame-pointer register, and a different return-address register. It is also a good idea to test with the return-value register being the same as one of the parameter registers. The pass should also be tested with just a few registers overall, which can be done by setting the helpers.ss variable `registers` to a shorter list. We suggest you don't modify helpers.ss but instead assign these variables in your driver file while testing.

This pass makes three transformations to achieve its effect. First, it converts the formal parameters of each `lambda` expression into locals and initializes these locals from the appropriate registers and frame locations.

```
(lambda (x0 ... xn-1 xn ... xn+m-1)
  (locals (local ...)
    body))
  ⇒  (lambda ()
       (locals (local ... rp x0 ... xn-1 xn ... xn+m-1)
         (begin
           (set! rp ra)
           (set! x0 p0)
           ...
           (set! xn-1 pn-1)
           (set! xn fv0)
           ...
           (set! xn+m-1 fvm-1)
           body)))
```

where *rp* is a fresh unique variable used to name the implicit return-address argument.

It assigns the register variables first to limit the live ranges of the *ra* and parameter registers.

For a `letrec` body, the transformation is a degenerate form of the transformation for `lambda` expressions, since a `letrec` body has no formal parameters. It does, still, have an implicit return-address argument.

(locals (*local* ...) *body*) ;== (locals (*local* ... *rp*) ;== (begin ;== (set! *rp ra*) ;== *body*)))

A *Body* helper used to implement `lambda` bodies can be used for `letrec` bodies as well if it receives an empty list of formal parameters for a `letrec` body.

Second, it assigns the appropriate registers and frame locations to the values of the actual parameters in each call. It replaces the arguments in the syntax for each call with a set of locations assumed to be live at the call, i.e., the return-address register *ra*, the frame-pointer register *fp*, and the set of locations into which the arguments have been placed. The order of the locations in this set is irrelevant, since it declares only the set of locations live at the point of the call.

```
(proc e0 ... en-1 en ... en+k-1)
  ⇒ (begin
      (set! fv0 en)
      ...
      (set! fvk-1 en+k-1)
      (set! p0 e0)
      ...
      (set! pn-1 en-1)
      (set! ra rp)
      (proc fp ra p0 ... pn-1 fv0 ... fvk-1))
```

Here, `rp` is the same unique variable chosen for the enclosing `lambda` expression.

We assign the parameter registers last to limit their live ranges.

Handling nontail calls requires a bit more work, but since our subset does not yet have nontail calls, we do not have to worry about them yet.

Third and finally, this pass converts each *Triv* or primitive call *tail* into an explicit assignment to the return-value register and a call to the return point. The return-value register is presumed live when the call is made, so it is listed as live at the point of call. The return-address register should not be included, since the last reference to it is in the call itself.

```
expr
  ⇒ (begin
      (set! rv expr)
      (rp fp rv))
```

Here again, `rp` is the same unique variable chosen for the enclosing `lambda` expression.

If a *Tail* expression is an `if` expression or `begin` expression, this pass should recur until it finds a *Tail* expression that is either a procedure call (in which case the second transformation above applies) or a *Triv* or primitive call (in which cases the third transformation applies).

5.5. `expose-frame-variable`

This pass should be updated to determine the actual frame-pointer register from the new helpers.ss variable `frame-pointer-register`. It should also use the helpers.ss `align-shift` variable to determine the amount by which frame indices need to be shifted to convert them into byte indices, to avoid hard-wiring the word-size into the compiler.

6. Boilerplate and Run-time Code

The boilerplate code does not change, but the frame-pointer, return-address, and return-value registers should be determined via the helpers.ss variables `frame-pointer-register`, `return-address-register`, and `return-value-register`. The helpers.ss procedure `emit-program` has been rewritten to do so.

The run-time code does not change.

7. Testing

A small set of invalid and valid tests for this assignment have been posted in tests6.ss. You should make sure that your compiler passes work at least on this set of tests.

8. Coding Hints

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

Use `make-begin` in each of the three passes to avoid nested `begin` expressions.