Assignment 14

Modified Sun Apr 26 09:33:49 EDT 2009

Contents

1. Background

This week we generalize the language by allowing

1. quoted list and vector constants;
2. `letrec` right-hand sides to be arbitrary expressions; and
3. variables to be assigned with `set!`.

2. Scheme Subset 14

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

```
Program  ⟶  Expr
Expr     ⟶  uvar
          |  (quote datum)
          |  (if Expr Expr Expr)
          |  (begin Expr* Expr)
          |  (lambda (uvar*) Expr)
          |  (let ([uvar Expr]*) Expr)
          |  (letrec ([uvar Expr]*) Expr)
          |  (set! uvar Expr)
          |  (prim Expr*)
          |  (Expr Expr*)
```

A *datum* is an *immediate*, pair of datums, or vector of datums; and an immediate is a boolean, a fixnum, or the empty list. The set of primitives and the constraints on unique variables and fixnums do not change.

3. Things to do

This week's passes must convert list and vector constants into code that explicitly allocates the specified lists and vectors (`convert-complex-datum`), locate assigned variables and allocate explicit locations for them (`uncover-assigned` and `convert-assignments`), and rewrite arbitrary Scheme `letrec` expressions to "pure" `letrec` expressions that bind only unassigned variables to `lambda` expressions (`purify-letrec`).

3.1. `verify-scheme`

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

3.2. `convert-complex-datum`

As we head toward UIL, which is independent of Scheme, we must at some point make explicit the representation of quoted pairs and vectors. These datatypes are specific to Scheme, although other languages may have similar datatypes, and in any case must ultimately be allocated as blocks of memory in the Scheme heap, just like pairs and vectors are allocated by `cons` and `make-vector`.

One way to accomplish this is to convert quoted pairs and vectors into code that builds them using `cons` and `make-vector`. Thus,

```
'(1 (2 (3 4 5)))
```

would be replaced by

```
(cons '1 (cons (cons '2 (cons (cons '3 (cons '4 (cons '5 '()))) '())) '()))
```

and

```
'#(32 (33 33) 34)
```

by

```
(let ([tmp.1 (make-vector '3)])
  (begin
    (vector-set! tmp '0 32)
    (vector-set! tmp '1 (cons 33 (cons 33 '())))
    (vector-set! tmp '2 34)
    tmp.1))
```

Once this transformation has been performed, `quote` expressions contain only immediate values, values that can be represented in a single machine word, i.e., fixnums, booleans, the empty list, and the void object.

We are not satisfied with merely replacing each complex constant with code that builds it, since this can cause the same, supposed constant, structure to be created multiple times, leading to less efficient code and sometimes surprising results. For example:

```
(let ([f.1 (lambda () '(1 . 2))])
  (eq? (f.1) (f.1)))
```

would become

```
(let ([f.1 (lambda () (cons '1 '2))])
  (eq? (f.1) (f.1)))
```

which creates the pair `(1 . 2)` once for each call to `f.1` and, thus, returns `#f` rather than `#t`. Either behavior is allowed by the Scheme standard, but it is preferable to avoid repeated allocation of the same constant structure.

Thus, we instead bind a new variable to the result of building the structure and reference the new variable in place of the quoted constant, as follows.

```
(let ([tmp.2 (cons '1 '2)])
  (let ([f.1 (lambda () tmp.2)])
    (eq? (f.1) (f.1))))
```

With this version, the pair is created only once, as desired.

This solution generalizes naturally to multiple complex constants, so that

```
(let ([f.1 (lambda () (cons '(1 . 2) '(3 . 4)))])
  (cons (f.1) (f.1)))
```

becomes

```
(let ([tmp.2 (cons '1 '2)] [tmp.3 (cons '3 '4)])
  (let ([f.1 (lambda () (cons tmp.2 tmp.3))])
    (cons (f.1) (f.1))))
```

`convert-complex-datum` should perform this transformation, wrapping the introduced `let` expression (if one is necessary) around the program as a whole.

It is possible but not necessary to commonize like structure that appears in two places. For example,

```
(eq? '(2) (cdr '(1 2)))
```

might become

```
(let ([tmp.1 (cons '2 '())] [tmp.2 (cons '1 (cons '2 '()))])
  (eq? tmp.1 (cdr tmp.2)))
```

and evaluate to `#f`, or it might become

```
(let ([tmp.1 (cons '2 '())])
  (let ([tmp.2 (cons '1 tmp.1)])
    (eq? tmp.1 (cdr tmp.2))))
```

and evaluate to `#t`.

The latter transformation is considerably more complex. To keep things simple, our implementation does the former transformation. Both are legal implementations of `quote` in Scheme.

3.3. `uncover-assigned`

Adding `set!` expressions is not a big deal, *per se*, but since the same variable can be accessed by more than one procedure, we need to place the values of each that is assigned in some location shared by the procedures, so that the effect of an assignment by one procedure can be observed by the others. Identifying the set of assigned variables is the first step in this process and is the job of `uncover-assigned`.

This pass locates all assigned variables and records each within the `lambda`, `let`, or `letrec` form that binds it. An assigned variable is one that appears somewhere within its scope on the left-hand side of a `set!` expression. To record this information, the body of each `lambda`, `let`, and `letrec` expression is wrapped in an `assigned` form listing the assigned variables bound by the expression. The `assigned` form is inserted even if the list of assigned variables is empty.

For example:

```
(let ([x.1 '0]) x.1)
```

becomes

```
(let ([x.1 '0])
  (assigned ()
    x.1))
```

since `x.1` is not assigned, while

```
(let ([x.1 '0])
  (begin
    (set! x.1 (+ x.1 '1))
    x.1))
```

becomes

```
(let ([x.1 '0])
  (assigned (x.1)
    (begin
      (set! x.1 (+ x.1 '1))
      x.1)))
```

since `x.1` is assigned, and

```
(let ([x.3 '10] [y.1 '11] [z.2 '12])
  (let ([f.9 (lambda (u.7 v.6)
               (begin
                 (set! x.3 u.7)
                 (+ x.3 v.6)))]
        [g.8 (lambda (r.5 s.4)
               (begin
                 (set! y.1 (+ z.2 s.4))
                 y.1))])
    (* (f.9 '1 '2) (g.8 '3 '4))))
```

becomes

```
(let ([x.3 '10] [y.1 '11] [z.2 '12])
  (assigned (x.3 y.1)
    (let ([f.9 (lambda (u.7 v.6)
                 (assigned ()
                   (begin
                     (set! x.3 u.7)
                     (+ x.3 v.6))))]
          [g.8 (lambda (r.5 s.4)
                 (assigned ()
                   (begin
                     (set! y.1 (+ z.2 s.4))
                     y.1)))])
      (assigned ()
        (* (f.9 '1 '2) (g.8 '3 '4))))))
```

since `x.3` and `y.1` are assigned but `z.2` is not; also `f.9` and `g.8` are not assigned; also `u.7` and `v.6` are not assigned; also `r.5` and `s.4` are not assigned.

Finally:

```
(let ([x.3 '10] [y.1 '11] [z.2 '12])
  (let ([f.7 '#f]
        [g.6 (lambda (r.5 s.4)
               (begin
                 (set! y.1 (+ z.2 s.4))
                 y.1))])
    (begin
      (set! f.7 (lambda (u.9 v.8)
                  (begin
                    (set! v.8 u.9)
                    (+ x.3 v.8))))
      (* (f.7 '1 '2) (g.6 '3 '4)))))
```

becomes

```
(let ([x.3 '10] [y.1 '11] [z.2 '12])
  (assigned (y.1)
    (let ([f.7 '#f]
          [g.6 (lambda (r.5 s.4)
                 (assigned ()
                   (begin
                     (set! y.1 (+ z.2 s.4))
                     y.1)))])
      (assigned (f.7)
        (begin
          (set! f.7
            (lambda (u.9 v.8)
              (assigned (v.8)
                (begin
                  (set! v.8 u.9)
                  (+ x.3 v.8)))))
          (* (f.7 '1 '2) (g.6 '3 '4)))))))
```

since `y.1` is assigned but `x.3` and `z.2` are not; also `f.7` is assigned but `g.6` is not; also `v.8` is assigned but `u.9` is not; also `r.5` and `s.4` are not assigned.

Determining at compile time whether a variable will be assigned at run time is generally undecidable, so we conservatively assume that a variable is assigned if it appears on the left-hand side of an assignment (`set!` expression). Thus, we consider `x.1` to be assigned in

```
(let ([x.1 '#f])
  (begin
    (if x.1 (set! x.1 '#t) (void))
    x.1))
```

even though it's possible to prove that the assignment to `x.1` is not reachable. It would be better to eliminate unreachable assignments in an earlier optimization pass in any case.

This pass may be implemented in a manner similar to `uncover-free`.

3.4. `purify-letrec`

This pass rewrites `letrec` expressions so that, in the output of the pass, all `letrec` expressions are "pure," i.e., bind only unassigned variables to `lambda` expressions.

The Scheme `letrec` expression

```
(letrec ((f (lambda (x) (+ x 1)))
         (g (lambda (y) (f (f y)))))
  (+ (f 1) (g 1)))
```

is pure, while

```
(letrec ((f (lambda (x) (+ x 1)))
         (g (lambda (y) (f (f y)))))
```

```
  (set! f (lambda (x) (- x 1)))
  (+ (f 1) (g 1)))
```

is impure, since `f` is assigned, and

```
(letrec ((f (cons (lambda (x) x) 17))
         (g (lambda (y) ((car f) ((car f) y)))))
  (+ ((car f) (cdr f)) (g 1)))
```

is impure, since the right-hand-side of the first binding is not a `lambda` expression.

Pure `letrec` expressions in the input should appear as pure `letrec` expressions in the output. In the intermediate language at the point where this pass occurs, pure `letrec` expressions do change slightly in that the unnecessary `assigned` form, which becomes superfluous because no `letrec`-bound variables are assigned after this pass, is dropped.

```
(letrec ((x e) ...) (assigned () body))  →
  (letrec ((x e) ...) body)
```

But what about impure `letrec` expressions?

The simplest solution is to replace all impure `letrec` expressions with the following standard `let`-and-`set!` expansion of `letrec`.

```
(letrec ((x e) ...) body)  →
  (let ((x (void)) ...)
    (let ((t e) ...)
      (set! x t) ...)
    body)
```

In our language, the transformation can be expressed as follows:

```
(letrec ((x e) ...)
  (assigned (x! ...)
    body))  →
  (let ((x (void)) ...)
    (assigned (x ...)
      (begin
        (let ((t e) ...)
          (assigned ()
            (begin (set! x t) ...)))
        body)))
```

where each *t* is a new variable.

Indeed, we could eliminate all `letrec` expressions in this manner, but leaving pure `letrec` expressions in the language allows subsequent passes to generate better code.

For that matter, we can do better even for some impure `letrec` expression, since some of the bindings of an impure `letrec` expression might be pure. The more sophisticated transformation described below produces fewer assigned variables and more (pure) `letrec` bindings.

For the more sophisticated transformation, handle each `letrec` expression

```
(letrec ([x e] ...) (assigned (x! ...) body))
```

as follows.

1. Transform the expressions *e* `...` and *body* via recursive application of this transformation to produce new *e* `...` and *body*.

2. Partition the bindings `[`x e`] ...` into three sets:

`[`x_s e_s`] ...`	*simple*
`[`x_l e_l`] ...`	*lambda*
`[`x_c e_c`] ...`	*complex*

3. Produce a set of nested `let` and `letrec` expressions from the partitioned bindings and *body* as shown below:

   ```
   (let ([xs es] ...)
     (assigned ()
       (let ([xc (void)] ...)
         (assigned (xc ...)
           (letrec ([xl el] ...)
             (let ([xt ec] ...)
               (assigned ()
                 (set! xc xt)
                 ...)
               body))))))
   ```

   where x_t `...` are fresh temporaries. Produce the innermost `let` only if `[`x_c e_c`] ...` is nonempty, and produce `begin` expressions where needed.

A binding `[`x e`]` is considered

simple if x is not assigned and e is a simple expression;

lambda if x is not assigned and e is a lambda expression; and

complex otherwise.

A simple expression contains no occurrences of the variables bound by the `letrec` expression and no applications unless nested within `lambda` expressions. The latter constraint prevents simple expressions from reaching a call to `call/cc` if we ever add `call/cc` to our language. Of course, at that time we would also rule out primitive calls to `call/cc` itself. It would also make sense to disallow `letrec` expressions, to prevent this pass from becoming nonlinear, and to disallow other expressions, such as `lambda` expressions, to reduce the cost of the "simple" check. Use your own judgement on this as long as you do treat as simple constants, references to variables not bound by the letrec, and primitive calls with simple operands.

3.5. `convert-assignments`

This pass eliminates `set!` expressions and, more importantly, assigned variables. To do so, it (a) introduces a `let` expression binding each assigned variable to a freshly allocated pair whose car is the original value of the variable, (b) replaces each reference to an assigned variable with a call to `car`, and (c) replaces each assignment with a call to `set-car!`. These transformations are described in more detail below.

a. Replace each `assigned` form

```
(assigned (x ...) expr)
```

with

```
(let ((x (cons t (void))) ...) expr)
```

and replace the binding of x by the enclosing `lambda` or `let` expression with an identical binding for t, where each t is a new unique name.

b. Replace each reference x to an assigned variable with

`(car` *x*`)`

c. Replace each assignment

`(set!` *x* *e*`)`

with

`(set-car!` *x* *e*`)`

For example,

```
(let ([x.3 '10] [y.1 '11] [z.2 '12])
  (assigned (x.3 z.2)
    (begin
      (set! x.3 (+ x.3 y.1))
      (set! z.2 (* y.1 '2))
      (cons y.1 z.2))))
```

becomes

```
(let ([x.5 '10] [y.1 '11] [z.4 '12])
  (let ([x.3 (cons x.5 (void))] [z.2 (cons z.4 (void))])
    (begin
      (set-car! x.3 (+ (car x.3) y.1))
      (set-car! z.2 (* y.1 '2))
      (cons y.1 (car z.2)))))
```

The code would still be correct Scheme code if we used the same names for the original and introduced bindings. We do the renaming to maintain the invariant that the same name is not used for two different variables. Renaming the original bindings, which are referenced only on the right-hand sides of the inserted `let`, is simpler than renaming the variables bound by the inserted `let`.

Although programs output from this pass no longer have `set!` expressions in them, the `set!` expressions have not really been eliminated—they've just changed form. For this reason, this transformation is referred to as "assignment conversion." In essence, assignment conversion makes explicit the locations where the values of assigned variables are stored.

A sufficient justification for `convert-assignments` is that it allows us to simplify the language a bit: we get to drop `set!` without adding anything new (we already have to handle `cons`, `car`, and `set-car!`). This is, however, only a superficial benefit. The real reason is that it makes explicit the locations where assigned variables are stored so that these locations can be shared by multiple procedures.

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 tests14.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.