MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall 2007

Recitation 24
Compilation

Compiling IFs

Here is the code from the explicit control evaluator for dealing with ifs.

```
ev-if
  (assign unev (reg exp))
  (save unev)
  (save env)
  (save continue)
  (assign continue (label ev-if-decide))
  (assign exp (op if-predicate) (reg unev))
  (goto (label eval-dispatch))
ev-if-decide
  (restore continue)
  (restore env)
  (restore unev)
  (test (op false?) (reg val))
  (branch (label ev-if-consequent))
ev-if-alternative
  (assign exp (op if-alternative) (reg unev))
  (goto (label eval-dispatch))
ev-if-consequent
  (assign exp (op if-consequent) (reg unev))
  (goto (label eval-dispatch))
```

A naive compiler would simply collect the used instructions from the corresponding portion of `ec-eval`. Compiling `(if a b c)` would thus yield:

```
1.    (save env)
2.    (save continue)
3.    (assign continue (label after-pred29))
4.    (assign val (op lookup-variable-value) (const a) (reg env))
5.    (goto (reg continue))
6.  after-pred29
7.    (restore continue)
8.    (restore env)
9.    (test (op false?) (reg val))
10.   (branch (label false-branch27))
11. true-branch28
12.   (assign val (op lookup-variable-value) (const b) (reg env))
13.   (goto (reg continue))
14. false-branch27
15.   (assign val (op lookup-variable-value) (const c) (reg env))
16.   (goto (reg continue))
```

The compiler has an advantage over `ec-eval` – it can keep track of which registers will be needed by an expression and which are changed, and eliminate many unnecessary operations. What lines can be removed from the above compilation to more efficiently handle ifs?

Decompiling Code - Short Examples

The following code results from the compilation of `(define x 3)`.

```
1.    (assign val (const 3))
2.    (perform (op define-variable!) (const x) (reg val) (reg env))
3.    (assign val (const ok))
4.    (goto (reg continue))
```

The following code results from the compilation of `(- x 2)`. For the evaluation of a combination, look for the procedure to be put into the proc register and the consing up of the argument list in the argl register. Are arguments evaluated left to right or right to left?

```
1.    (assign proc (op lookup-variable-value) (const -) (reg env))
2.    (assign val (const 2))
3.    (assign argl (op list) (reg val))
4.    (assign val (op lookup-variable-value) (const x) (reg env))
5.    (assign argl (op cons) (reg val) (reg argl))
6.    (test (op primitive-procedure?) (reg proc))
7.    (branch (label primitive-branch14))
8.  compiled-branch13
9.    (assign continue (label after-call12))
10.   (assign val (op compiled-procedure-entry) (reg proc))
11.   (goto (reg val))
12. primitive-branch14
13.   (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
14. after-call12
15.   (goto (reg continue))
```

The following code results from the compilation of `(if a b c)`. When you see labels like true-branch and false-branch, you should think if.

```
1.    (assign val (op lookup-variable-value) (const a) (reg env))
2.    (test (op false?) (reg val))
3.    (branch (label false-branch16))
4.  true-branch17
5.    (assign val (op lookup-variable-value) (const b) (reg env))
6.    (goto (label after-if15))
7.  false-branch16
8.    (assign val (op lookup-variable-value) (const c) (reg env))
9.  after-if15
10.   (goto (reg continue))
```

The following code results from the compilation of `(lambda (x) x)`. Note that the body of the lambda (lines 3-7) are not evaluated. The label is just saved along with the current environment to make the procedure object.

```
1.   (assign val (op make-compiled-procedure) (label entry19) (reg env))
2.   (goto (label after-lambda18))
3. entry19
4.   (assign env (op compiled-procedure-env) (reg proc))
5.   (assign env (op extend-environment) (const (x)) (reg argl) (reg env))
6.   (assign val (op lookup-variable-value) (const x) (reg env))
7.   (goto (reg continue))
8. after-lambda18
9.   (goto (reg continue))
```

What scheme expression led to the following compiled code?

```
 1.    (assign val (op make-compiled-procedure) (label entry2) (reg env))
 2.    (goto (label after-lambda1))
 3.  entry2
 4.    (assign env (op compiled-procedure-env) (reg proc))
 5.    (assign env (op extend-environment) (const (a)) (reg argl) (reg env))
 6.    (save continue)
 7.    (save env)
 8.    (assign proc (op lookup-variable-value) (const >) (reg env))
 9.    (assign val (const 0))
10.    (assign argl (op list) (reg val))
11.    (assign val (op lookup-variable-value) (const a) (reg env))
12.    (assign argl (op cons) (reg val) (reg argl))
13.    (test (op primitive-procedure?) (reg proc))
14.    (branch (label primitive-branch11))
15.  compiled-branch10
16.    (assign continue (label after-call9))
17.    (assign val (op compiled-procedure-entry) (reg proc))
18.    (goto (reg val))
19.  primitive-branch11
20.    (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
21.  after-call9
22.    (restore env)
23.    (restore continue)
24.    (test (op false?) (reg val))
25.    (branch (label false-branch4))
26.  true-branch5
27.    (assign val (op lookup-variable-value) (const a) (reg env))
28.    (goto (reg continue))
29.  false-branch4
30.    (assign proc (op lookup-variable-value) (const -) (reg env))
31.    (assign val (op lookup-variable-value) (const a) (reg env))
32.    (assign argl (op list) (reg val))
33.    (test (op primitive-procedure?) (reg proc))
34.    (branch (label primitive-branch8))
35.  compiled-branch7
36.    (assign val (op compiled-procedure-entry) (reg proc))
37.    (goto (reg val))
38.  primitive-branch8
39.    (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
40.    (goto (reg continue))
41.  after-call6
42.  after-if3
43.  after-lambda1
44.    (perform (op define-variable!) (const f) (reg val) (reg env))
45.    (assign val (const ok)))
46.    (goto (reg continue))
```

Lines 8–20:

Lines 30–40:

Lines 8–40:

Lines 1–46:

Another example of a compiled expression: what was the scheme expression for this compiled code? (SICP Ex. 5.35)

```
  (assign val (op make-compiled-procedure) (label entry16)
                                           (reg env))
  (goto (label after-lambda15))
entry16
  (assign env (op compiled-procedure-env) (reg proc))
  (assign env
          (op extend-environment) (const (x)) (reg argl) (reg env))
  (assign proc (op lookup-variable-value) (const +) (reg env))
  (save continue)
  (save proc)
  (save env)
  (assign proc (op lookup-variable-value) (const g) (reg env))
  (save proc)
  (assign proc (op lookup-variable-value) (const +) (reg env))
  (assign val (const 2))
  (assign argl (op list) (reg val))
  (assign val (op lookup-variable-value) (const x) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch19))
compiled-branch18
  (assign continue (label after-call17))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch19
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
after-call17
  (assign argl (op list) (reg val))
  (restore proc)
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch22))
compiled-branch21
  (assign continue (label after-call20))
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch22
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
after-call20
  (assign argl (op list) (reg val))
  (restore env)
  (assign val (op lookup-variable-value) (const x) (reg env))
  (assign argl (op cons) (reg val) (reg argl))
  (restore proc)
  (restore continue)
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-branch25))
compiled-branch24
  (assign val (op compiled-procedure-entry) (reg proc))
  (goto (reg val))
primitive-branch25
  (assign val (op apply-primitive-procedure) (reg proc) (reg argl))
  (goto (reg continue))
after-call23
after-lambda15
  (perform (op define-variable!) (const f) (reg val) (reg env))
  (assign val (const ok))
```