Brandeis University

COMPUTER SCIENCE (1)21B (SPRING TERM, 2018)

Structure and Interpretation of Computer Programs

Compiling for a Register Machine

Here is a compiler similar in spirit to the "metacircular compiler" we have already described in a previous handout, except that the code is more like that for a *register machine*. It has four registers: an environment register `*ENV*`, a continuation register `*CNT*`, a function register `*FUN*`, and a value register `*VAL*`. Each is initialized to `#F`. When the compiled code for a Scheme expression is evaluated (i.e., run), it leaves its value in the value register.

The machine has the following instructions and expressions:

`(set!` ⟨*register*⟩ ⟨*value*⟩`)` Self-explanatory.

`(continue)` Call the continuation (using contents of the continuation register). This is like a "restore" or "return from subroutine call."

`(invoke)` Do a procedure call (using register contents).

```
(make-procedure ...)
```

```
(make-continuation ...)
```

The important feature of this compiler is that during the running of compiled code, information is passed via the registers of the machine, rather than through the parameter passing mechanism of the Scheme language. (Recall in the previous compiler that `(lambda (x) ...)` in the source language was simulated by `(lambda (x k) ...)` in the target language.)

Thus compiled code is always of the form `(lambda () ...)`, where the body contains the code—we use `lambda` only to *delay* the evaluation of a piece of code, and not to pass parameters.

When a piece of compiled code is run, it always leaves its value in the `*VAL*` register. The code terminates by calling the continuation, which says what to do next. The code thus leaves the continuation **unchanged.**

A procedure consists of code (of the body of the procedure), and an environment.

```
(define (make-procedure code env)
  (vector 'procedure code env))
```

Here's how we *implement* our machine. A procedure is invoked as follows: the procedure to be applied is in the function register; the argument is in the value register. Set the environment register to the environment mentioned in the procedure; run the code in the procedure.

```
(define (invoke)
  ;; invoke what's in the *FUN* register
  ;; argument is in the *VAL* register
  (if (not (eq? 'procedure (vector-ref *FUN* 0)))
      (error "Bad procedure"))
  (let ((code (vector-ref *FUN* 1)))
    (set! *ENV* (cons *VAL* (vector-ref *FUN* 2)))
    (code)))
```

Note that `(make-procedure ...)` and `(invoke)` then become expressions in our (target) register machine language.

A continuation consists of code (to be run later), and the contents of the function, environment, and continuation register. (Thus the continuation can be thought of as a stack.)

```
(define (make-continuation code fun env cnt)
  (vector 'continuation code fun env cnt))
```

When a value is returned by running a piece of code, the code terminates by calling the continuation. This calling resets the function, environment, and continuation registers, and then runs the code saved in the continuation.

The continuation register stores pieces of code (and associated environments) that are to be executed later. There is an important **invariant** that describes the functioning of the continuation:

Suppose we execute an `invoke` *with appropriate contents loaded into the* `*FUN*` *and* `*VAL*` *registers; denote the contents of the* `*CNT*` *register at that moment as* κ*. That is,* κ *describes what to do once the function application is completed, and the resultant value is in the* `*VAL*` *register. The code generated by the compiler guarantees that when that value is indeed (later) in the* `*VAL*` *register, the* `*CNT*` *register once again has value* κ*. The continuation could have* changed *during the function evaluation, but will return to its original value at the conclusion of that evaluation.*

How do we prove *this? By induction!*

```
(define (continue)
  ;; continue with what's in the *CNT* register
  ;; returned value is in the *VAL* register
  (if (not (eq? 'continuation (vector-ref *CNT* 0)))
      (error "Bad continuation"))
  (let ((code (vector-ref *CNT* 1)))
    (set! *FUN* (vector-ref *CNT* 2))
    (set! *ENV* (vector-ref *CNT* 3))
    (set! *CNT* (vector-ref *CNT* 4))
    (code)))
```

```
(define (compile exp env-names)
  (cond ((constant? exp) (compile-constant exp))
        ((variable? exp)
         (compile-variable exp env-names))
        ((letrec? exp)
         (compile-letrec (letrec-var exp)
                         (letrec-val exp)
                         (letrec-body exp)
                         env-names))
        ((lambda? exp)
         (compile-lambda (binder exp)
                         (body exp)
                         env-names))
        ((if? exp)
         (compile-if (predicate exp)
                     (then-part exp)
                     (else-part exp)
                     env-names))
        ((sequence? exp)
         (compile-sequence (cdr exp)
                           env-names))
        ((enter? exp)
         (compile-enter (enter-body exp)
                        env-names))
        ((exit? exp)
         (compile-exit (exit-body exp)
                       env-names))
        (else ; it's an application!
         (compile-application (function-of exp)
                              (argument-of exp)
                              env-names))))
```

```
(define (compile-constant c)
  (lambda ()
    (set! *VAL* c)
    (continue)))

(define (compile-variable v env-names)
  (let ((i (lookup-variable v env-names)))
    (lambda ()
      (set! *VAL* (list-ref *ENV* i))
      (continue))))

(define (compile-if test then else env-names)
  (let ((test-code (compile test env-names))
        (then-code (compile then env-names))
        (else-code (compile else env-names)))
    (lambda ()
      (set! *CNT* (make-continuation
                   (lambda ()
                     ((if *VAL* then-code else-code)))
                   #F
                   *ENV*
                   *CNT*))
      (test-code))))

(define (compile-sequence sequence env-names)
  (let ((compiled-first
         (compile (car sequence) env-names)))
    (if (null? (cdr sequence))
        compiled-first
        (let ((compiled-rest
               (compile-sequence (cdr sequence) env-names)))
          (lambda ()
            (set! *CNT* (make-continuation
                         compiled-rest #F *ENV* *CNT*))
            (compiled-first))))))
```

```
(define (compile-application fun arg env-names)
  (let ((fun-code (compile fun env-names))
        (arg-code (compile arg env-names)))
    (lambda ()
      (set! *CNT* (make-continuation
                   (lambda ()
                     (set! *CNT* (make-continuation
                                  invoke
                                  *VAL*  ; we're saving the function!
                                  *ENV*
                                  *CNT*))
                     (arg-code) ; when this code is run, the argument
                                ; is left in the *VAL* register--
                                ; when we call (continue), the saved
                                ; function is restored to the *FUN*
                                ; register, prior to calling (invoke)
                     )
                   #F
                   *ENV*
                   *CNT*))
      (fun-code) ; when this code is run, the function is
                 ; left in the *VAL* register
      )))

(define (compile-lambda v exp env-names)
  (let ((body-code (compile exp (cons v env-names))))
    (lambda ()
      (set! *VAL* (make-procedure body-code *ENV*))
      (continue))))
```

```
(define (compile-letrec var val body env-names)
  (let ((new-env-names (cons var env-names)))
    (let ((val-code (compile val new-env-names))
          (body-code (compile body new-env-names)))
      (lambda ()
        (set! *ENV* (cons '*UNDEFINED* *ENV*))
        (set! *CNT* (make-continuation
                     (lambda ()
                       (set-car! *ENV* *VAL*)
                       (body-code))
                     #F
                     *ENV*
                     *CNT*))
        (val-code)))))

(define (compile-enter exp env-names)
  (let ((body-code
         (compile exp (cons '*EXIT* env-names))))
    (lambda ()
      (set! *ENV* (cons *CNT* *ENV*))
      (body-code))))

(define (compile-exit exp env-names)
  (let ((body-code (compile exp env-names))
        (i (lookup-variable '*EXIT* env-names)))
    (lambda ()
      (set! *CNT* (list-ref *ENV* i))
      (body-code))))
```

```
; Stuff for primitive procedures, odds and ends

(define (unop op)
  (make-procedure (lambda ()
                    (set! *VAL* (op (car *ENV*)))
                    (continue))
                  '()))

(define (binop op)
  (let ((step2
         (lambda ()
           (set! *VAL* (op (cadr *ENV*) (car *ENV*)))
           (continue))))
    (make-procedure
      (lambda ()
        (set! *VAL* (make-procedure step2 *ENV*))
        (continue))
      '())))

(define initial-continuation
  (make-continuation
    (lambda () *VAL*)
    #F
    #F
    #F))

(define initial-compile-time-environment
  '(+ - * 1+ cons car cdr nil true false))

(define (try exp)
  (compile exp initial-compile-time-environment))
```

Next, you will see compiled code that is produced by a *modified compiler* that does not generate procedures that can be run, but *list structures* that can be read by you. The benefit to the gentle reader is that the entire piece of compiled code can be surveyed at once. (Of course, we could build an interpreter for evaluating this code.) The biggest change is that `(lambda () ⟨code⟩)` is just changed to ⟨*code*⟩.

```
(try '320)
((set! *val* 320) (continue))
;No value

(try '1+)
((set! *val* (list-ref *env* 3)) (continue))
;No value

(try '(if true 0 1))
((set!
  *cnt*
  (make-continuation
   (if *val*
    (begin (set! *val* 0) (continue))
    (begin (set! *val* 1) (continue)))
   ()
   *env*
   *cnt*))
 (set! *val* (list-ref *env* 11))
 (continue))
;No value

(try '(lambda (x) x))
((set!
  *val*
  (make-procedure
    ((set! *val* (list-ref *env* 0)) (continue))
    *env*))
 (continue))
;No value
```

```
(try '(lambda (x) (lambda (y) x)))
((set!
  *val*
  (make-procedure
   ((set!
     *val*
     (make-procedure ((set! *val* (list-ref *env* 1))
                      (continue))
                     *env*))
    (continue))
   *env*))
 (continue))
;No value

(try '(begin 0 1 2))
((set!
  *cnt*
  (make-continuation
   ((set! *cnt* (make-continuation
                 ((set! *val* 2) (continue))
                 ()
                 *env*
                 *cnt*))
    (set! *val* 1)
    (continue))
   ()
   *env*
   *cnt*))
 (set! *val* 0)
 (continue))
;No value
```

```
(try '(1+ 0))
((set!
  *cnt*
  (make-continuation
   ((set! *cnt* (make-continuation
                  invoke *val* *env* *cnt*))
    (set! *val* 0)
    (continue)))
  ()
  *env*
  *cnt*)
 (set! *val* (list-ref *env* 3))
 (continue))
;No value

(try '(enter (1+ (exit 0))))
((set! *env* (cons *cnt* *env*))
 (set!
  *cnt*
  (make-continuation
   ((set! *cnt* (make-continuation
                  invoke *val* *env* *cnt*))
    (set! *cnt* (list-ref *env* 0))
    (set! *val* 0)
    (continue)))
  ()
  *env*
  *cnt*)
 (set! *val* (list-ref *env* 4))
 (continue))
```

QUESTION: Suppose we compiled the Scheme code for the metacircular evaluator (given as a list structure, of course). We'd get back compiled code for... what? Or what if we compiled the code for the *compiler? What do these thought experiments suggest as a way of* implementing software?