CS450 - Structure of Higher Level Languages

The Explicit-Control Evaluator

December 7, 2020

The Explicit-Control Evaluator

- We have shown how simple scheme programs can be transformed into register machines.
- We will now perform this transformation on a more complex program, the metacircular evaluator
- The explicit-control evaluator shows how the underlying procedure-calling and argument-passing mechanisms used by the evaluator can be described in terms of operations on registers and stacks.
- the explicit-control evaluator can serve as an implementation of a Scheme interpreter, written in a language very similar to the native machine language of conventional computers.
- The evaluator can be executed by the register-machine simulator.
- It can also be the basis for building a hardware implementation!

The Explicit-Control Evaluator

- We must specify the operations to be used in our register machine.
- We described the metacircular evaluator in terms of abstract syntax, using procedures such as `quoted?` and `make-procedure`.
- In implementing the register machine, we could expand these procedures into sequences of elementary list-structure memory operations, and implement them on our register machine.
- However, this would make our evaluator very long, obscuring the basic structure with details.
- For clarity, we will include some procedures as primitives.

The Explicit-Control Evaluator

- Our Scheme evaluator register machine includes a stack and seven registers:
  1. `exp` is used to hold the expression to be evaluated
  2. `env` contains the environment in which the evaluation is to be performed
  3. `val` contains the value obtained by evaluating the expression in the designated environment at the end of an evaluation
  4. `continue` is used to implement recursion. The evaluator needs to call itself recursively, since evaluating an expression requires evaluating its subexpressions.
  5. `proc`, `argl`, and `unev` are used in evaluating combinations.

eval-dispatch

- `eval-dispatch` corresponds to the eval procedure of the metacircular evaluator.
- When the controller starts at `eval-dispatch`, it evaluates the expression specified by `exp` in the environment specified by `env`.
- When evaluation is complete, the controller will go to the entry point stored in `continue`, and the `val` register will hold the value of the expression.
- The structure of `eval-dispatch` is a case analysis on the syntactic type of the expression to be evaluated

eval-dispatch

```
eval-dispatch
  (test (op self-evaluating?) (reg exp))
  (branch (label ev-self-eval))
  (test (op variable?) (reg exp))
  (branch (label ev-variable))
  (test (op quoted?) (reg exp))
  (branch (label ev-quoted))
  (test (op assignment?) (reg exp))
  (branch (label ev-assignment))
  (test (op definition?) (reg exp))
  (branch (label ev-definition))
  (test (op if?) (reg exp))
  (branch (label ev-if))
  (test (op lambda?) (reg exp))
  (branch (label ev-lambda))
  (test (op begin?) (reg exp))
  (branch (label ev-begin))
  (test (op application?) (reg exp))
  (branch (label ev-application))
  (goto (label unknown-expression-type))
```

Evaluating Simple Expressions

```
ev-self-eval
  (assign val (reg exp))
  (goto (reg continue))
ev-variable
  (assign val (op lookup-variable-value) (reg exp) (reg env))
  (goto (reg continue))
ev-quoted
  (assign val (op text-of-quotation) (reg exp))
  (goto (reg continue))
ev-lambda
  (assign unev (op lambda-parameters) (reg exp))
  (assign exp (op lambda-body) (reg exp))
  (assign val (op make-procedure)
              (reg unev) (reg exp) (reg env))
  (goto (reg continue))
```

`ev-lambda` uses `unev` and `exp` to hold the parameters and body of the lambda expression so that they can be passed to `make-procedure`, along with `env`.

Evaluating Procedure Applications

- A procedure application is specified by a combination of an operator and operands.
- The operator is a subexpression whose value is a procedure, and the operands are subexpressions whose values are the arguments.
- The metacircular eval recursively evaluates each element of the combination, and then passing the results to apply, which performs the actual procedure application.
- The explicit-control evaluator does the same thing; these recursive calls are implemented by goto instructions, with use of the stack to save registers that will be restored after the recursive call returns.
- Before each call we will be careful to identify which registers must be saved (because their values will be needed later).

Evaluating Procedure Applications

- We begin the evaluation of an application by evaluating the operator to produce a procedure, which will later be applied to the evaluated operands.
- We move the operator to the `exp` register and go to `eval-dispatch`.
- The environment in the env register is already the correct one in which to evaluate the operator.
- However, we save env because we will need it later to evaluate the operands.
- We extract the operands into `unev` and save this on the stack.
- We set up continue so that `eval-dispatch` will resume at `ev-appl-did-operator` after the operator has been evaluated.
- First, however, we save the old value of continue, which tells the controller where to continue after the application.

Evaluating Procedure Applications

```
ev-application
  (save continue)
  (save env)
  (assign unev (op operands) (reg exp))
  (save unev)
  (assign exp (op operator) (reg exp))
  (assign continue (label ev-appl-did-operator))
  (goto (label eval-dispatch))
```

Evaluating Procedure Applications

At this point the operator is evaluated, we move on to evaluate the operands.

```
ev-appl-did-operator
  (restore unev)   ; the operands
  (restore env)
  (assign argl (op empty-arglist))
  (assign proc (reg val))  ; the operator
  (test (op no-operands?) (reg unev))
  (branch (label apply-dispatch))
  (save proc)
```

Evaluating Procedure Applications

- Each cycle of the argument-evaluation loop evaluates an operand from `unev` and accumulates the result into `argl`.
- We place the operand in the `exp` register and go to `eval-dispatch`, after setting continue so that execution will resume with the argument-accumulation phase.
- When an operand has been evaluated, the value is accumulated into the list held in `argl`.
- It is then removed from `unev`, and the argument-evaluation continues.
- A special case is made for the evaluation of the last operand, which is handled at ev-appl-last-arg.
- In this case we don't need to save `unev` and the environment.

Evaluating Procedure Applications

```
ev-appl-operand-loop
  (save argl)
  (assign exp (op first-operand) (reg unev))
  (test (op last-operand?) (reg unev))
  (branch (label ev-appl-last-arg))
  (save env)
  (save unev)
  (assign continue (label ev-appl-accumulate-arg))
  (goto (label eval-dispatch))
```

Evaluating Procedure Applications

```
ev-appl-accumulate-arg
  (restore unev)
  (restore env)
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (assign unev (op rest-operands) (reg unev))
  (goto (label ev-appl-operand-loop))

ev-appl-last-arg
  (assign continue (label ev-appl-accum-last-arg))
  (goto (label eval-dispatch))
ev-appl-accum-last-arg
  (restore argl)
  (assign argl (op adjoin-arg) (reg val) (reg argl))
  (restore proc)
  (goto (label apply-dispatch))
```

Applying a Procedure

Test whether it is a primitive or a user-defined procedure.

```
  apply-dispatch
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
  (branch (label compound-apply))
  (goto (label unknown-procedure-type))

primitive-apply
  (assign val (op apply-primitive-procedure)
              (reg proc)
              (reg argl))
  (restore continue)
  (goto (reg continue))
```

Applying a Procedure

Test whether it is a primitive or a user-defined procedure.

```
apply-dispatch
  (test (op primitive-procedure?) (reg proc))
  (branch (label primitive-apply))
  (test (op compound-procedure?) (reg proc))
  (branch (label compound-apply))
  (goto (label unknown-procedure-type))

primitive-apply
  (assign val (op apply-primitive-procedure)
              (reg proc)
              (reg argl))
  (restore continue)
  (goto (reg continue))
```

Applying a Procedure

```
compound-apply
  (assign unev (op procedure-parameters) (reg proc))
  (assign env (op procedure-environment) (reg proc))
  (assign env (op extend-environment)
              (reg unev) (reg argl) (reg env))
  (assign unev (op procedure-body) (reg proc))
  (goto (label ev-sequence))
```

Sequences and Tail Recursion

- `ev-sequence` is analogous to the metacircular evaluator's `eval-sequence` procedure.
- It handles sequences of expressions in procedure bodies or in explicit begin expressions.
- begin expressions are evaluated by placing the sequence of expressions to be evaluated in `unev`, saving `continue` on the stack, and jumping to `ev-sequence`.
- You should understand how sequences work if you choose to do question 2 in HW9.

```
ev-begin
  (assign unev (op begin-actions) (reg exp))
  (save continue)
  (goto (label ev-sequence))
```

Evaluating Sequences

```
ev-sequence
  (assign exp (op first-exp) (reg unev))
  (test (op last-exp?) (reg unev))
  (branch (label ev-sequence-last-exp))
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))
ev-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-sequence))
ev-sequence-last-exp
  (restore continue)
  (goto (label eval-dispatch))
```

Evaluating Sequences

- The code forms a loop where each expression is being evaluated.
- If there are more expressions after this one, they are saved to `unev`, and the environment is saved to `env`.
- The register `continue` tells us where to go after evaluation, and then `eval-dispatch` is called.
- When returning to the sequence, `env` and `unev` are restored, the rest of the expressions are stored in `unev` and the loop continues.
- The value of the whole sequence is the value of the last expression, so after evaluating the last expression we only need to continue at the entry point currently held on the stack.
- This makes the evaluator tail recursive, because nothing is left on the stack from the sequence after evaluating the last expression.

Disabling Tail Recursion

- We can disable tail recursion by making a small change to the `ev-sequence` process.
- As it is now, all but the last expression are treated the same: We are saving the registers, evaluating the expression, returning to restore the registers, and repeating this until all the expressions have been evaluated.
- Making the last expression do the same will make us come back after evaluating the last expression and undo the register saves, effectively making tail recursion functioning as regular recursion.

Evaluating Sequences

```
ev-sequence
  (test (op no-more-exps?) (reg unev))
  (branch (label ev-sequence-end))
  (assign exp (op first-exp) (reg unev))
  (save unev)
  (save env)
  (assign continue (label ev-sequence-continue))
  (goto (label eval-dispatch))
ev-sequence-continue
  (restore env)
  (restore unev)
  (assign unev (op rest-exps) (reg unev))
  (goto (label ev-sequence))
ev-sequence-end
  (restore continue)
  (goto (reg continue))
```

Evaluating Conditionals

```
 ev-if
  (save exp) ; save expression for later
  (save env)
  (save continue)
  (assign continue (label ev-if-decide))
  (assign exp (op if-predicate) (reg exp))
  (goto (label eval-dispatch))  ; evaluate the predicate

ev-if-decide
  (restore continue)
  (restore env)
  (restore exp)
  (test (op true?) (reg val))
  (branch (label ev-if-consequent))
```

Evaluating Conditionals

```
ev-if-alternative
  (assign exp (op if-alternative) (reg exp))
  (goto (label eval-dispatch))
ev-if-consequent
  (assign exp (op if-consequent) (reg exp))
  (goto (label eval-dispatch))
```

Notice that for the cond question in the HW you'll have to go to `ev-sequence` (why?)

Evaluating Definitions

```
 ev-definition
  (assign unev (op definition-variable) (reg exp))
  (save unev) ; save variable for later
  (assign exp (op definition-value) (reg exp))
  (save env)
  (save continue)
  (assign continue (label ev-definition-1))
  (goto (label eval-dispatch))
ev-definition-1
  (restore continue)
  (restore env)
  (restore unev)
  (perform
   (op define-variable!) (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue))
```

Evaluating Assignments

```
 v-assignment
  (assign unev (op assignment-variable) (reg exp))
  (save unev) ; save variable for later
  (assign exp (op assignment-value) (reg exp))
  (save env)
  (save continue)
  (assign continue (label ev-assignment-1))
  (goto (label eval-dispatch))
ev-assignment-1
  (restore continue)
  (restore env)
  (restore unev)
  (perform
  (op set-variable-value!) (reg unev) (reg val) (reg env))
  (assign val (const ok))
  (goto (reg continue))
```

Monitoring the Evaluator

- Stack monitoring allows us to follow the behavior of the evaluator.
- The monitored stack keeps track of its max depth and number of pushes.
- Useful for Q3-5 of HW9.

```
print-result
 (perform (op print-stack-statistics))
 (perform
  (op announce-output) (const ";;; EC-Eval value:"))
 ... ; same as before
```

Monitoring the Evaluator – Example

```
;;; EC-Eval input:
(define (factorial n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
(total-pushes = 3 maximum-depth = 3)
;;; EC-Eval value:
ok
;;; EC-Eval input:
(factorial 5)
(total-pushes = 144 maximum-depth = 28)
;;; EC-Eval value:
120
```