CS450 - Structure of Higher Level Languages

Spring 2018 – The Explicit-Control Evaluator

April 25, 2018
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!
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.
Our Scheme evaluator register machine includes a stack and seven registers:

1. \texttt{exp} is used to hold the expression to be evaluated.
2. \texttt{env} contains the environment in which the evaluation is to be performed.
3. \texttt{val} contains the value obtained by evaluating the expression in the designated environment at the end of an evaluation.
4. \texttt{continue} is used to implement recursion. The evaluator needs to call itself recursively, since evaluating an expression requires evaluating its subexpressions.
5. \texttt{proc}, \texttt{arg1}, and \texttt{unev} are used in evaluating combinations.
• 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
  (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.
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).
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.
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))
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)
```
Each cycle of the argument-evaluation loop evaluates an operand from \( \text{unev} \) and accumulates the result into \( \text{argl} \). We place the operand in the \( \text{exp} \) register and go to \( \text{eval-dispatch} \), 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 \( \text{argl} \). It is then removed from \( \text{unev} \), and the argument-evaluation continues.

A special case is made for the evaluation of the last operand, which is handled at \( \text{ev-appl-last-arg} \). In this case we don’t need to save \( \text{unev} \) and the environment.
ev-appl-operand-loop
  (save arg1)
  (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))
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))
compound-apply
   (assign unev (op procedure-parameters) (reg proc))
   (assign env (op procedure-environment) (reg proc))
   (assign env (op extend-environment)
      (reg unev) (reg arg1) (reg env))
   (assign unev (op procedure-body) (reg proc))
   (goto (label ev-sequence))