CS6848 - Principles of Programming Languages

Principles of Programming Languages

V. Krishna Nandivada

IIT Madras

Outline

Data Types and their Representations

- Want to define new data types.
  - a specification - tells us what data (and what operations on that data) we are trying to represent.
  - implementation - tells us how we do it.
- We want to arrange things so that you can change the implementation without changing the code that uses the data type (user = client; implementation = supplier/server).
- Both the specification and implementation have to deal with two things: the data and the operations on the data.
- Vital part of the implementation is the specification of how the data is represented. We will use the notation $\lceil v \rceil$ for "the representation of data 'v'.

Numbers

- **Data specification**: Non negative numbers.
- **operations**:

$$
\begin{array}{lcl}
\texttt{zero} & = & \lceil 0 \rceil \\
(\texttt{is-zero?}\ \lceil n \rceil) & = & \begin{cases} \texttt{\#t} & n = 0 \\ \texttt{\#f} & n \neq 0 \end{cases} \\
(\texttt{succ}\ \lceil n \rceil\ ) & = & \lceil n+1 \rceil \\
(\texttt{pred}\ \lceil n+1 \rceil\ ) & = & \lceil n \rceil
\end{array}
$$

- **Extensions to do other operations**: Should work irrespective of the underlying representation.

```
(define plus
  (lambda (x y)
    (if (is-zero? x) y
        (succ (plus (pred x) y)))))
```

  - Irrespective of the representation $(\texttt{plus}\ \lceil x \rceil \lceil y \rceil) = \lceil x+y \rceil$

Scheme Representation of Numbers

$\lceil n \rceil$ = the Scheme integer n

```
(define zero 0)
(define is-zero? zero?)
(define succ (lambda (n) (+ n 1)))
(define prec (lambda (n) (- n 1)))
```


Unary Representation of Numbers

$\lceil 0 \rceil$ ()
$\lceil n+1 \rceil$ = `( cons #t` $\lceil n \rceil$`)`

- So the integer n is represented by a list of n `#t`'s.
- Satisfy the specification:

```
(define zero = '())

(define is-zero? null?)

(define succ
  (lambda (n) (cons #t n)))

(define pred cdr)
```


Data Representation (contd). Example 2: Finite functions

- **Data specification**: a function whose domain is a finite set of Scheme symbols, and whose range is unspecified.
- **Specification of operation**: *Aka - the interface*

$$\begin{array}{lcl} \texttt{empty-ff} & = & \lceil \phi \rceil \\ (\texttt{apply-ff}\ \lceil f \rceil\ s) & = & f(s) \\ (\texttt{extend-ff}\ s\ v\ \lceil f \rceil) & = & \lceil g \rceil \end{array}$$

$$\text{where } g(s') = \begin{cases} v & s' = s \\ f(s') & \text{Otherwise} \end{cases}$$

- Interface gives the type of each procedure and a description of the intended behavior of each procedure.

Procedural Representation

$f = \lceil \{(s_1, v_1), ..., (s_n, v_n)\} \rceil$ iff $(f\ s_i) = v_i$.
Implement the operations by:

```
(define apply-ff
  (lambda (ff z) (ff z)))

(define empty-ff
  (lambda (z)
    (error 'env-lookup
          (format "couldn't find ~s" z))))

(define extend-ff
  (lambda (key val ff)
    (lambda (z)
      (if (eq? z key)
        val
        (apply-ff ff z)))))
```

Procedural Representation

Examples

```
> (define ff-1 (extend-ff 'a 1 empty-ff))
> (define ff-2 (extend-ff 'b 2 ff-1))
> (define ff-3 (extend-ff 'c 3 ff-2))
> (define ff-4 (extend-ff 'd 4 ff-3))
> (define ff-5 (extend-ff 'e 5 ff-4))
> ff-5
<Procedure>
> (apply-ff ff-5 'd)
4
> (apply-ff empty-ff 'c)
error in env-lookup: couldn't find c.
> (apply-ff ff-3 'd)
error in env-lookup: couldn't find d.
>(define ff-new (extend-ff 'd 6 ff-4))
> (apply-ff ff-new 'd)
> 6
```


Association-list Representation

$\lceil\{(s_1, v_1), ..., (s_n, v_n)\}\rceil = ((s_1.v_1)...(s_n.v_n))$

```
(define empty-ff '())

(define extend-ff
  (lambda (key val ff)
    (cons (cons key val) ff)))

(define apply-ff
  (lambda (alist z)
    (if (null? alist)
      (error 'env-lookup
             (format "couldn't find ~s" z))
      (let ((key (caar alist))
            (val (cdar alist))
            (ff (cdr alist)))
        (if (eq? z key) val (apply-ff ff z))))))
```


Association-list Representation

Examples

```
> (define ff-1 (extend-ff 'a 1 empty-ff))
> (define ff-2 (extend-ff 'b 2 ff-1))
> (define ff-3 (extend-ff 'c 3 ff-2))
> (define ff-4 (extend-ff 'd 4 ff-3))
> ff-4
((d . 4) (c . 3) (b . 2) (a . 1))
> (apply-ff ff-4 'd)
4
```

Useless Assignment: Specification and Implementation of Stack as a type.

Outline

1. Extending the Scheme language
   - Data types
2. Interpreters
   - Stack machine
   - Environments for an interpreter
   - Cells for Variables
   - Closures
   - Recursive environments
   - Interpreting MicroJava

Interpreters

The complexity of Interpreters depend on the language under consideration.

- Simple/Complex
- Environments
- Cells
- Closures
- Recursive Environments

Stack machine grammar

- **Goal**: interpreter for a stack machine.
- The machine will have two components: an action and a stack.
- The stack contains the data in the machine.
- We will represent the stack as a list of Scheme values, with the top of the stack at the front of the list.
- The action represents the instruction stream being executed by the machine.
-
```
Action ::= halt
         | incr; Action
         | add; Action;
         | push Integer ; Action
         | pop; Action
```
- Our interpreter - `eval-action`: takes an action and a stack and returns the value produced by the machine at the completion of the action.
- Convention: the machine produces a value by leaving it on the top of the stack when it halts.

Specification of Operations

Specification for `eval-action`. *Our VM*

- What `(eval-action a s)` does for each possible value of `a`.

```
(eval-action halt s) = (car s)

(eval-action incr; a (v w ...)) =
          (eval-action a (v+1 w ...))

(eval-action add; a (v w x ...)) =
          (eval-action a ((v+w) x ...))

(eval-action push v; a (w ...)) =
          (eval-action a (v w ...))

(eval-action pop; a (v w ...)) =
          (eval-action a (w ...))
```

- Is the specification complete? How to prove the same?

Representation of Operations

To write Scheme code to implement the specification of `eval-action`, we need to specify a representation of the type of actions. (*Our bytecode*).

- A simple choice - use lists.

$$\begin{array}{lcl} \lceil halt \rceil & = & (\mathrm{halt}) \\ \lceil incr; a \rceil & = & (\mathrm{incr}\ .\ \lceil a \rceil) \\ \lceil add; a \rceil & = & (\mathrm{add}\ .\ \lceil a \rceil) \\ \lceil push\ v; a \rceil & = & (\mathrm{push\ v}\ .\ \lceil a \rceil) \\ \lceil pop; a \rceil & = & (\mathrm{pop}\ .\ \lceil a \rceil) \end{array}$$

- An action is represented as a list of instructions.
- Typical action is `(push 3 push 4 add halt)`

A Stack Machine Interpreter

```
(define eval-action
  (lambda (action stack)
    (let ((op-code (car action)))
      (case op-code
        ((halt)
         (car stack))
        ((incr)
         (eval-action (cdr action)
           (cons (+ (car stack) 1) (cdr stack))))
        ((add)
         (eval-action (cdr action)
           (cons (+ (car stack) (cadr stack)) (cddr stack))))
        ((push)
         (let ((v (cadr action)))
           (eval-action (cddr action) (cons v stack))))
        ((pop)
         (eval-action (cdr action) (cdr stack)))
        (else
          (error 'eval-action "unknown op-code:" op-code))))))
```


Interpreter in action

Running the Interpreter

```
> (define start
   (lambda (action)
     (eval-action action '())))

> (start '(push 3 push 4 add halt))
7
```


Outline

Interpreters (contd.): Environment

- An environment is a finite function - that maps identifiers to values.
- Why do we need an environment?
- **Specification**:

$$\begin{array}{lcl} \texttt{empty-Env} & = & \lceil\phi\rceil \\ (\texttt{apply-Env}\ \lceil f\rceil\ s) & = & f(s) \\ (\texttt{extend-Env}\ s\ v\ \lceil f\rceil) & = & \lceil g\rceil \\ & & \text{where } g(s') = \begin{cases} v & s' = s \\ f(s') & \text{Otherwise} \end{cases} \end{array}$$

Environment implementation

```
(define empty-env
  (lambda () '()))

(define extend-env
  (lambda (id val env)
    (cons (cons id val) env)))


(define apply-env
  (lambda (env id)
    (if (or (null? env) (null? id))
      null
      (let ((key (caar env))
            (val (cdar env))
            (env-prime (cdr env)))
        (if (eq? id key) val (apply-env env-prime id))))))

(define extend-env-list
  (lambda (ids vals env) ... )
```


extend-env-list

```
(define extend-env-list
  (lambda (ids vals env)
    (if (null? ids)
        env
        (extend-env-list
          (cdr ids)
          (cdr vals)
          (extend-env (car ids) (car vals) env)))))
```

Home reading: Read Scheme alist representation and see how the above routines can be compacted.

Interpreter with environment

```
(define eval-Expression
  (lambda (Expression)
    (record-case Expression
      ...
      (PlusExpression  (Tkn1 Tkn2 Expression1 Expression2 Tkn3)
        (+ (eval-Expression Expression1)
           (eval-Expression Expression2))))

      (Identifier (Token) (apply-env env Token))
     ... ))

(define run
  (lambda ()
       (record-case root
          (Goal (Expression Token)
          (eval-Expression Expression (empty-env)))
    (else (error 'run ``Goal not found'')))))
```


Extending an environment - let expression

```
(LetExpression (Token1 Token2 Token3
                          List Token4 Expression Token5)
  (let* ((ids  (get-ids List))
         (exps (get-exprs List))
         (vals (map (lambda (Expression)
                      (eval-Expression Expression env))
                     exps))
         (new-env (extend-env-list ids vals env)))
    (eval-Expression Expression new-env)))
```

```
> (map cdr '((1 2 3) (3 4 5)))
((2 3) (4 5))
```

Useless assignment: How to interpret Let*?

Outline

1. Extending the Scheme language
   - Data types
2. Interpreters
   - Stack machine
   - Environments for an interpreter
   - Cells for Variables
   - Closures
   - Recursive environments
   - Interpreting MicroJava

Update to variables

- One undesirable feature of Scheme: assignment to variables.
- A variable has a name and address where it stores the value, which can be updated.

```
(define make-cell
  (lambda (value)
    (cons '*cell value)))

(define deref-cell cdr)

(define set-cell! set-cdr!)
```

- When we extend an environment, we will create a cell, store the initial value in the cell, and bind the identifier to the cell.

```
(define extend-env
  (lambda (id value env)
    (cons (id (make-cell value)) env)))
```


Example: Interpreting a let expression

```
(let ((x 7))
  (+ (let ((y x)
        (x (+ 2 x)))
       (* x y)) x)
```


Outline

1. Extending the Scheme language
   - Data types
2. Interpreters
   - Stack machine
   - Environments for an interpreter
   - Cells for Variables
   - Closures
   - Recursive environments
   - Interpreting MicroJava

Include headers in Scheme

```
(load "recscm.scm")
(load "records")
(load "tree")
```


Closures

To represent user-defined procedures, we will use closures.

```
(define-record closure (formals body env))
```


Closures

```
(define eval-Expression
  (lambda (Expression env)
    (record-case Expression
     ...
     (ProcedureExp (Token1 Token2 Token3
                    List Token4 Expression Token5)
       (make-closure List Expression env))
     (Application (Token1 Expression List Token2)
       (let*
         ((clos (eval-Expression Expression env))
          (ids  (get-formals clos))
          (vals (map (lambda (Exp)
                       (eval-Expression Exp env))
                     List))
          (static-env (get-closure-env clos))
          (new-env
            (extend-env-list ids vals static-env)))
          (body (get-body clos))
        (eval-Expression body new-env)))
     ...)))
```


If Stmt

```
(define eval-Expression
  (lambda (Expression env)
    (record-case Expression
     ...

     (IfExpression (Token1 Token2 Expression1
                    Expression2 Expression3 Token3)
       (if (eval-Expression Expression1 env)
           (eval-Expression Expression2 env)
           (eval-Expression Expression3 env)
     ...)))
```

Outline

Recursive Environments for recursive definitions

- We need two kinds of environment records.
  - Normal environments contain cells.
  - A recursive environment contains a RecDeclarationList. If one looks up a recursively-defined procedure, then it gets closed in the environment frame that contains it:

```
(define-record normal-env (ids vals env))

(define-record rec-env (recdecl-list env))

(define eval-Expression
  (lambda (Expression env)
    (record-case Expression
      ...
      (RecExpression (Token1 Token2 Token3
                      List Token4 Expression Token5)
        (eval-Expression
          Expression
          (make-rec-env List env)))
      (else (error ...)))))
```


Recursive environments

```
(define apply-env
 (lambda (env id)
   (record-case env
     ...
     (rec-env (recdecl-list old-env)
       (let ((id-list (get-ids recdecl-list)))
         (if (member? id id-list)
           (let* ((RecProc (get-decl id recdecl-list))
                  (ProcExpr (get-proc-expr RecProc)))

             (make-cell (make-closure  ;; a cell
                  (get-formals ProcExpr)
                  (get-body ProcExp) env)))
           (apply-env old-env id)))))))
```


Microjava

- Evaluating a method, env - stmt List; return expr
- Evaluating a stmt, env - a switch case.
- Evaluating an Expr, env - a switch case.
- Inheritance
- Finding classes.
- Finding Methods
- Finding variables.
- Allocating Objects