Imperative Programming

Handout #38

CS251 Lecture 30
April 20, 2005

Franklyn Turbak

Wellesley College

Functional vs. Imperative Programming

- Functional Programming (e.g., OCaml, Scheme, Haskell)
  - Heavy use of first-class functions
  - Immutablility /persistence: variables and data structures do not change over time.
  - Expressions denote values
- Imperative Programming (e.g., C, Pascal, Fortran, Ada ; core of Java, C++)
  - Mutability/side effects: variables, data structures, procedures, input/output streams can change over time:
  - Often a distinction between expressions (which denote values) and statements (which perform actions). In some languages, expressions do both.
  - Imperative languages often have non-local control flow features (gotos , non-local exits, exceptions). We will study these later.
- Combining functional and imperative programming
  - OCaml and Scheme do have imperative features, but used sparingly. They are “mostly-functional” languages.
  - First-class functions + side effects are at the core of many important programming idioms. E.g., can simulate Java-like object-oriented.

HOILEC = HOFL + Explicit Mutable Cells

HOILEC	Specification	OCAML
`(cell` E`)`	Return a cell whose contents is the value of E	`ref` E
`(^` E`)`	Return current contents of the cell designated by E.	`!` E
`(:=` E_{cell} E_{new}`)`	Change contents of the cell designated by E_{cell} to be the value of E_{new}. Returns the old contents of E_{cell}.	E_{cell} `:=` E_{new} returns unit, not old value
`(cell=?` E_1 E_2`)`	Test if E_1 and E_2 denote the same cell.	E_1 `=` E_2
`(cell?` E`)`	Test if E denotes a cell.	N/A
`(println` E`)`	Display string rep. of E value followed by newline; return value.	`(print_string (... ^ "\n"))` (returns unit value)

Sequential Execution

In the presence of side effects, order of evaluation is important! HOILEC sequentializes via

`(seq` E_1 ... E_n`)`
Evaluate $E_1 \ldots E_n$ in order and return the value of E_n.

Notes:

- `(seq` E_1 ... E_n`)` can be considered sugar for `(bindseq ((`I_1 E_1`)` ... `(`I_n E_n`))` I_n`);` I_i `fresh`.
- HOILEC `(seq` E_1 ... E_n`)` correspsonds to:
  OCaml's `(`E_1`;` ... `;` E_n`)`
  Scheme's `(begin` E_1 ... E_n`)`
  Java and C's $\{$$E_1$`;` ... `;` E_n`;`$\}$ (no value returned)

Mutable Cell Example

```
(bind a (cell (+ 3 4))
  (seq (println (^ a))
       (:= a (* 2 (^ a)))
       (println (^ a))
       (:= a (+ 1 (^ a)))
       (println (^ a))
       (bind b (cell (^ a))
         (bind c b
           (seq (println (cell=? a b))
                (println (cell=? b c))
                (:= c (/ (^ c) 5))
                (println (^ a))
                (println (^ b))
                (^ c))))))
```

Imperative Factorial in Java

```
public static int fact (int n) {
  int ans = 1;
  while (n > 0) {
    // Order of assignments is critical!
    ans = n * ans ;
    n = n - 1;
    return ans ;
  }
}
```

Imperative Factorial in HOILEC

```
(def (fact n)
  (bindpar ((num (cell n))
            (ans (cell 1)))
    (bindrec
      ((loop (fun ()
               (if (= (^ num) 0)
                   (^ ans)
                   (seq
                     (:= ans (* (^ num) (^ ans)))
                     (:= num (- (^ num) 1))
                     (loop))))))
      (loop))))
```

Mutable Stacks in HOILEC

```
(def (make-stack) (cell #e))
(def (stack-empty? stk) (empty? (^ stk)))
(def (top stk) (head (^ stk)))
(def (push! val stk)
  (:= stk (prep val (^ stk))))
(def (pop! stk)
  (bind t (top stk)
    (seq (:= stk (tail (^ stk)))
         t)))
; Example
(bind s (make-stack)
  (seq (push! 2 s) (push! 3 s) (push! 5 s)
       (+ (pop! s) (pop! s))))
```

HOILEC Argument Evaluation Order

Operand expressions to primitive and function applications in HOILEC are evaluated left to right:

```
(bind c (cell 1)
  (+ (seq (:= c (* 10 (^ c)))
          (^ c))
     (seq (:= c (+ 2 (^ c)))
          (^ c))))
```

Listing `fib` args in HOILEC

```
(hoilec (x) (list (fib x) (^ args))
  (def args (cell #e))
      ;; collects args to fib (in reverse)
  (def (fib n)
    (seq (:= args (prep n (^ args)))
         (if (<= n 1)
             n
             (+ (fib (- n 1)) (fib (- n 2)))))))
```

Listing `fib` args in HOFL

Without mutable cells, need to "thread" state through computation:

```
(hofl (x) (fib x #e)
  (def (fib n args) ; Returns list of
                    ;   (1) fib and
                    ;   (2) args
    (if (<= n 1)
        (list n (prep n args))
        (bind ans1 (fib (- n 1) (prep n args))
          (bind ans2 (fib (- n 2) (nth 2 ans1))
            (list (+ (nth 1 ans1) (nth 1 ans2))
                  (nth 2 ans2)))))))
```

Maintaining State in HOILEC functions

The following `fresh` function (similar to OCaml's `StringUtils.fresh`) illustrates how HOILEC functions can maintain state in a local environment.

```
(def fresh
  (bind count (cell 0)
    (fun (s)
      (bind n (^ count)
        (seq (:= count (+ n 1))
             (str+ (str+ s ".")
                   (toString n)))))))
```

Promises in HOILEC

- `(delayed` E_{thunk}`)` Return a promise to evaluate the thunk (nullary function) denoted by E_{thunk} at a later time.
- `(force` $E_{promise}$`)` If the promise denoted by $E_{promise}$ has not yet been evaluated, evaluate it and remember and return its value. Otherwise, return the remembered value.

Example:

```
(bind inc! (bind c (cell 0)
             (fun() (seq (:= c (+ 1 (^ c)))
                         (^ c))))
  (bind p (delayed (fun () (println (inc!))))
    (+ (force p) (force p))))
```

HOIL-EC Promise Implementation 1

```
(def (delayed thunk)
  (list thunk (cell #f) (cell #f)))

(def (force promise)
  (if (^ (nth 2 promise))
      (^ (nth 3 promise))
      (bind val ((nth 1 promise)) ;  dethunk !
        (seq (:= (nth 2 promise) #t)
             (:= (nth 3 promise) val)
             val))))
```

HOILEC Promise Implementation 2

```
(def (delayed thunk)
  (bindpar ((flag (cell #f))
            (memo (cell #f)))
    (fun ()
      (if (^ flag)
          (^ memo)
          (seq (:= memo (thunk)) ; dethunk!
               (:= flag #t)
               (^ memo))))))

(def (force promise) (promise))
```

HOILIC = HOFL + Implicit Mutable Cells

HOILIC is a version of of HOFL in which:

- All variables I are bound to cells.
- Variable references I denote the current contents of the associated cell.
- `(<-` I E_{new}`)` changes the contents of the cell designated by I to be the value of E_{new} and returns old contents.

Example: `(bindpar ((a 2) (b 3)) (seq  (<- a (+ a b)) a))`

Similar to Other Languages:

Scheme: `(let ((a 2) (b 3)) (begin (set! a (+ a b)) a))`

Java/C: `int a = 2; int b = 3; a = a + b;` *use a*

Pascal:
```
begin var a: int := 2;
      var b: int := 3;
      a := a + b; use a end
```

Object-Oriented Programming (OOP) Example

```
public class MyPoint {
  private static numPoints = 0; // class variable
  private int x, y; // instance variables

  public MyPoint (int ix, int iy) {// constructor method
    numPoints++; // count the points we’ve made.
    x = ix; // initialize coordinates
    y = iy;}

  public static int count () {// class method
    return numPoints;}

  // Instance methods
  public int getX () {return x;}
  public int setX (int newX) {x = newX;}
  public int getY () {return y;}
  public int setY (int newy) {y = newY;}
  public int translate (int dx, int dy) {
    this.setX(x + dx); // Use setX and setY to illustrate "this"
    this.setY(y + dy);}
}
```

OOP Example, Part 2

Sample use of `MyPoint` class:

```
// Code using MyPoint (in some other class)
public static int testMyPoint () {
  MyPoint p1 = new MyPoint(3,4);
  MyPoint p2 = new MyPoint(5,6);
  p1.setX(p2.getY()); // sets x of p1 to 6
  p2.setY(MyPoint.count()); // sets y of p2 to 2
  p1.translate(1,2); // sets x of p1 to 7 and y of p1
  return (1000 * p1.getX())
         + (100 * p1.getY())
         + (100 * p2.getX())
         + p2.getY(); // returns 7652
}
```

OOP in HOILIC, Part 1

```
(def test-my-point
  (fun ()
    (bindseq ((p1 ((my-point "new") 3 4))
              (p2 ((my-point "new") 5 6)))
      (seq
        ((p1 "set-x") ((p2 "get-y")))
        ((p2 "set-y") ((my-point "count")))
        ((p1 "translate") 1 2)
        (+ (* 1000 ((p1 "get-x")))
           (+ (* 100 ((p1 "get-y")))
              (+ (* 10 ((p2 "get-x")))
                 ((p2 "get-y"))))))))))
```

OOP in HOILIC, Part 2

```
(def my-point
  (bind num-points 0 ; class variable
    (fun (cmsg) ; class message
     (cond
       ((str= cmsg "count") (fun () num-points)) ; Act like class method
       ((str= cmsg "new") ; Act like constructor method
        (fun (ix iy)
          (bindpar ((x 0) (y 0)) ; instance variables
            (seq (<- num-points (+ num-points 1)) ; count points
                 (<- x ix) (<- y iy)
                 (bindrec ; create and return instance dispatcher function.
                   ((this ; Give the name "this" to instance dispatcher
                     (fun (imsg) ; instance message
                       (cond ((str= imsg "get-x") (fun () x))
                             ((str= imsg "get-y") (fun () y))
                             ((str= imsg "set-x") (fun (new-x) (<- x new-x)))
                             ((str= imsg "set-y") (fun (new-y) (<- y new-y)))
                             ((str= imsg "translate)
                              (fun (dx dy) (seq ((this "set-x") (+ x dx))
                                                ((this "set-y") (+ y dy)))))
                             (else "error: unknown instance message")))))
                   this))))) ; Return instance dispatcher as result of "new"
       (else "error: unknown class message")))))
```

Other Mutable Structures

- In addition to ref cells, OCaml supports arrays with mutable slots. But all variables and list nodes are immutable!
- Scheme has mutable list node slots (changed via `set-car!` & `set-cdr!`) and vectors with mutable slots (modified via `vector-set!`).
- C and Pascal support mutable records and array variables, which can be stored either on the stack or on the heap. Stack-allocated variables are sources of big headaches (we shall see this later).
- Almost every language has stateful Input/Outptut (I/O) operations for reading from/writing to files.

Advantages of Side Effects

- Can maintain and update information in a modular way. Examples:
  - Report the number of times a function is invoked. Much easier with cells than without!
  - Using `StringUtils.fresh` to generate fresh names – avoids threading name generator throughout entire mini-language implementation.
  - Tracing functions in OCaml.
- Computational objects with local state are nice for modeling the real world. E.g., gas molecules, digital circuits, bank accounts

Disadvantages of Side Effects

- Lack of referential transparency makes reasoning harder.

  Referential transparency: evaluating the same expression in the same environment always gives the same result.

  In language without side effects, `(+` E E`)` can always be safely transformed to `(* 2` E`)`. But not true in the presence of side effects! E.g. E `= (seq (:= c (+ (^ c) 1)) a)`.

  Even in a purely functional call-by-value language, non-termination is a kind of side effect. Are the following HOILEC expressions always equal?

  ```
  (if E1 E2 E3)
    <=?=> (bind I E3 (if E1 E2 I)) ; I fresh
  ```

- Aliasing makes reasoning in the presence of side effects particularly tricky. E.g. HOILEC example:

  ```
  (+ (^ a) (seq (:= b (+ 1 (^ b))) (^ a))
     <=?=>  (seq  (:= b (+ 1 (^ b))) (* 2 (^ a)))
  ```

- Harder to make persistent structures (e.g., aborting a transaction, rolling back a database to a previous saved point).