CS251 Programming Languages **Handout # 3**
Prof. Lyn Turbak **May 4, 2016**
Wellesley College

Imperative and Object-Oriented Programming with Implicit Cells (HOILIC)

We have introduced imperative programming in the context of HOILEC, a language with explicit cells. In HOILEC, all variables have *immutable* bindings to values, but one of the values is a *mutable* explicit cell. OCAML is a real-world language that uses this model of state.

However, in most real-world languages with imperative and/or object-oriented features (e.g., C, C++, JAVA, JAVASCRIPT, PYTHON, ADA, PASCAL, and even RACKET and COMMON LISP), all variables have *mutable* bindings to values. In these languages, each variable names an *implicit* cell whose contents can change over time.

For example, here are imperative versions of the factorial function written in C and in RACKET:

```
// C version of imperative factorial
int fact (int n) {
  int ans = 1;
  while (n > 0) {
    ans = n*ans;
    n = n-1;
  }
  return ans;
}

;; Scheme version of imperative factorial
(define (fact n)
  (let ((ans 1))
    (letrec ((loop (lambda ()
                     (if (<= n 0)
                         ans
                         (begin (set! ans (* n ans))
                                (set! n (- n 1))
                                (loop))))))
      (loop))))
```

In both examples, the variables `n` and `ans` name implicit cells with time-varying integer contents. The contents of an implicit cell are accessed simply by refering to the variable name (which implicitly **dereferences** the cell — i.e., extracts its contents). The contents of an implicit cell are changed by performing an **assignment** (written Id_{var} `=` E_{newval} in C and `(set!` Id_{var} E_{newval}`)` in RACKET).

In this handout, we explore imperative programming with implicit cells in the context of the mini-language HOILIC = HOFL + Implicit Cells.

1 HOILIC Overview

HOILIC is like HOILEC except for the following differences:

- Every variable in HOILIC names an implicit cell. In HOILIC, the contents of a cell can be changed by the assignment expression `(<-` Id_{var} E_{newval}`)`. Evaluating this expression (1) replaces the contents of the implicit cell named by Id_{var} with the value of the expression E_{newval} and (2) returns the *previous* contents of Id_{var}. For example:[1]

[1] The call-by-value HOILIC interpreter uses the prompt `hoilic-cbv` to distinguish it from the interpreters for versions of HOILIC that use other paramter-passing mechanisms.

```
hoilic-cbv> (def a 17)
a

hoilic-cbv> (def b a)
b

hoilic-cbv> (list a b)
(list 17 17)

hoilic-cbv> (<- a 42)
17

hoilic-cbv> (list a b)
(list 42 17)

hoilic-cbv> (<- b (<- a b)) ; Swaps contents of vars a and b
17

hoilic-cbv> (list a b)
(list 17 42)
```

- Unlike HOILEC, HOILIC does *not* include explicit cell values or primitive operations on these values. The reason is that explicit cells are easy to construct in a language with implicit cells (see PS9).

- In HOILIC, the `bindrec` construct can be expressed as syntactic sugar rather than as a kernel construct:

  ```
  (bindrec ((Id1 E1) ... (Idn En)) Ebody)
    ⇝ (bindpar ((Id1 (sym *undefined*)) ... (Idn (sym *undefined*)))
         (seq (<- Id1 E1)
                 ⋮
              (<- Id1 En)
              Ebody))
  ```

  Not only does this guarantee that the identifiers $Id_1 \ldots Id_n$ are defined in a single mutual recursive scope, but it also allows the expression E_i to directly reference the identifiers $Id_1 \ldots Id_{i-1}$. (In HOFL and HOILEC, any such references would denote "black holes".) For example, the expression

  ```
  (bindrec ((a 1)
            (f (fun () (seq (<- a (* a 10)) a)))
            (b (* 2 a))
            (c (f))
            (d (+ (* 3 a) (+ (* 4 (f)) (* 5 a)))))
    (list a b c d))
  ```

  evaluates to the value `(list 100 2 10 930)`.

  If the E_i directly references any identifiers $Id_i \ldots Id_n$, these will appear to have the value `(sym *undefined*)`. For example,

  ```
  (bindrec ((a (+ 1 2))
            (b (list a b c))
            (c (* 4 a)))
    (list a b c))
  ```

  has the value `(list 3 (list 3 (sym *undefined*) (sym *undefined*)) 12)` and

```
(bindrec ((a (+ 1 2))
          (b (- c a))
          (c (* 4 a)))
  (list a b c))
```

signals an error, because it is not able to subtract 3 from `(sym *undefined*)`.

In all other respects, HOILIC is like HOILEC. In particular, HOILIC includes HOILEC's syntactic sugar constructs `(seq` E_1 ... E_n`)` and `(while` E_{test} E_{body}`)`.

2 HOILIC Examples

This section presents HOILIC verisons of several examples we considered earlier in the context of HOILEC.

2.1 Factorial

```
(def (fact n)
  (bind ans 1
    (seq (while (> n 0)
           (seq (<- ans (* ans n))
                (<- n (- n 1))))
         ans)))

hoilic-cbv> (fact 4)
24

hoilic-cbv> (fact 5)
120
```

2.2 Fresh Variables

```
(def fresh
  (bind count 0
    (fun (s)
      (str+ (str+ s ".")
            (toString (<- count (+ count 1)))))))

hoilic-cbv> (fresh "a")
"a.0"

hoilic-cbv> (fresh "b")
"b.1"

hoilic-cbv> (fresh "a")
"a.2"
```

2.3 Promises

```
(def (make-promise thunk)
  (bindpar ((flag #f)
            (memo #f))
    (fun ()
      (if flag
          memo
          (seq (<- flag #t)
```

```
                (<- memo (thunk))
                memo)))))

(def (force promise) (promise))

hoilic-cbv> (def p (make-promise (fun () (println (+ 1 2)))))
p

hoilic-cbv> (* (force p) (force p))
3
9
```

2.4 Message-Passing Stacks

```
(def (new-stack)
  (bind elts (empty)
    ;; Dispatch function representing stack instance
    (fun (msg)
      (cond
        ((str= msg "empty?") (empty? elts))
        ((str= msg "push")
         (fun (val)
           (seq (<- elts (prep val elts))
                val))) ; Return pushed val
        ((str= msg "top")
         (if (empty? elts)
             (error "Attempt to top an empty stack!" elts)
             (head elts)))
        ((str= msg "pop")
         (if (empty? elts)
             (error "Attempt to pop an empty stack!" elts)
             (bind result (head elts)
               (seq (<- elts (tail elts))
                    result))))
        (else (error "Unknown stack message:" msg))
        ))))

hoilic-cbv> (def s1 (new-stack))
s1

hoilic-cbv> (def s2 (new-stack))
s2

hoilic-cbv> ((s1 "push") 17)
17

hoilic-cbv> ((s1 "push") 42)
42

hoilic-cbv> ((s1 "push") 23)
23

hoilic-cbv> (while (not (s1 "empty?"))
              (println ((s2 "push") (s1 "pop"))))
23
42
17
#f
```

```
hoilic-cbv> (while (not (s2 "empty?"))
              (println (s2 "pop")))
17
42
23
#f

hoilic-cbv> (s2 "top")
EvalError: Hoilic Error -- Attempt to top an empty stack!:#e
```

2.5 Message-Passing Points

```
(def my-point
  (bind num-points 0 ; class variable
    (fun (cmsg) ; class message
     (cond
       ((str= cmsg "count") num-points) ; acts like a class method
       ((str= cmsg "new") ; acts like a constructor method
        (fun (ix iy)
          (bindpar ((x ix) (y iy)) ; instance variables
            (seq (<- num-points (+ num-points 1)) ; count points
                 (bindrec ; create and return instance dispatcher
                    function.
                   ((this ; gives the name "this" to instance =
                      instance method dispatcher
                     (fun (imsg) ; instance message
                       (cond
                         ;; the following are instance methods
                         ((str= imsg "get-x") x)
                         ((str= imsg "get-y") 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)))))
                         ((str= imsg "toString")
                          (str+ "<"
                                (str+ (toString x)
                                      (str+ ","
                                            (str+ (toString y)
                                                  ">")))))
                         (else "error: unknown instance message"
                            imsg)))))
                    this))))) ;  return instance as the result of "new"
       (else "error: unknown class message" cmsg)
       ))))

hoilic-cbv> (def p1 ((my-point "new") 3 4))
p1

hoilic-cbv> (def p2 ((my-point "new") 5 6))
p2

hoilic-cbv> (list (p1 "toString") (p2 "toString"))
(list "<3,4>" "<5,6>")
```

```
hoilic-cbv> ((p1 "set-x") (p2 "get-y"))
3

hoilic-cbv> ((p2 "set-y") (my-point "count"))
6

hoilic-cbv> (list (p1 "toString") (p2 "toString"))
(list "<6,4>" "<5,2>")

hoilic-cbv> ((p1 "translate") 7 8)
4

hoilic-cbv> (list (p1 "toString") (p2 "toString"))
(list "<13,12>" "<5,2>")
```