HOILEC: Imperative Programming with Explicit Cells

This is a second draft of a handout with parts that still need to be fleshed out.

Thus far our focus has been on the function-oriented programming paradigm (also known as the functional programming paradigm), which is characterized by the following:

- heavy use of first-class functions
- immutability/persistence: variables and data structures do not change over time
- expressions denote values.

Ocaml, Scheme, and Haskell are exemplars of this paradigm, though only Haskell enforces immutability, making it a purely functional language. Because Ocaml and Scheme support some mutability features, they are sometimes called mostly functional languages.

We now begin to explore the imperative programming paradigm, which is characterized by the following features:

- mutability/side effects: variables, data structures, procedures, and input/output streams can change over time.
- 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.

Imperative languages include C, Ada, Pascal and Fortran. Imperative programming is also the foundation for object-oriented languages like Java and C++.

We will study imperative programming by extending HoFL with some imperative features. We will see that mixing imperative features with HoFL’s first-class functions is a powerful combination that can express many important programming idioms, such as memoization and object-oriented programming. Such idioms are used extensively in real-world function-oriented languages that support imperative features (e.g., Ocaml and Scheme).

1 HOILEC = HoFL + Explicit Mutable Cells

We begin our exploration of imperative programming by extending HoFL with a new kind of value: the mutable cell. This is a one-slot data structure whose value can change over time. We christen the resulting language HOILEC = Higher-Order Imperative Language with Explicit Cells.

Fig. 1 summarizes the new primitive operations in HOILEC. This includes operations for creating mutable cells (cell), getting the current value in a mutable cell (*), changing the value in a mutable cell (:=), testing the equality of two mutable cells (cell=), and determining if a value is a cell (cell?). The new primitive operations also include print and println for displaying values.

Here are some examples involving the operators:
<table>
<thead>
<tr>
<th>HOILEC</th>
<th>Specification</th>
<th>OCAML</th>
</tr>
</thead>
<tbody>
<tr>
<td>(cell $E$)</td>
<td>Return a cell whose contents is the value of $E$</td>
<td>ref $E$</td>
</tr>
<tr>
<td>(“$E$”)</td>
<td>Return current contents of the cell designated by $E$.</td>
<td>! $E$</td>
</tr>
<tr>
<td>($:= E_{cell} E_{new}$)</td>
<td>Change contents of the cell designated by $E_{cell}$ to be the value of $E_{new}$. Returns the old contents of $E_{cell}$.</td>
<td>$E_{cell} := E_{new}$ (this returns unit, not the old value)</td>
</tr>
<tr>
<td>(cell= $E_1$ $E_2$)</td>
<td>Test if $E_1$ and $E_2$ denote the same cell.</td>
<td>$E_1 = E_2$</td>
</tr>
<tr>
<td>(cell? $E$)</td>
<td>Test if $E$ denotes a cell.</td>
<td>N/A</td>
</tr>
<tr>
<td>(print $E$)</td>
<td>Displays the string representation of the value of $E$ and returns the value.</td>
<td>(print_string ... ) (this returns unit, not the value)</td>
</tr>
<tr>
<td>(println $E$)</td>
<td>Displays the string representation of the value of $E$ followed by newline and returns the value.</td>
<td>(print_string (... ^ &quot;\n&quot;)) (this returns unit, not the value)</td>
</tr>
</tbody>
</table>

Figure 1: New primitive operations added to HOFL to yield HOILEC.

hoilec> (def a (cell 3))
a
hoilec> (^ a)
3
hoilec> (def b (cell 3))
b
hoilec> (^ b)
3
hoilec> (:= a 17)
3
hoilec> (list (^ a) (^ b))
(list 17 3)
hoilec> (cell= a b)
#f
hoilec> (cell= a a)
#t
hoilec> (cell? a)
#t
hoilec> (cell? (^ a))
#f
hoilec> (println (+ 1 2))
3
hoilec> (print (+ 1 2))
33
It turns out that OCAML is similar to HOILEC because it also provides state-based computation via mutable cells. Fig. 1 shows the OCAML cell operations corresponding to the HOILEC ones.

In the presence of side effects, order of evaluation is important! HOILEC provides sequential evaluation via the following construct:

\[(\text{seq } E_1 \ldots E_n)\]

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

This need not be a new kernel construct because it can be implemented by the following desguaring:

\[(\text{seq } E_1 \ldots E_n) \Rightarrow (\text{bindseq } ((I_1 E_1) \ldots (I_n E_n)) I_n); I_i \text{ fresh}\]

HOILEC’s \((\text{seq } E_1 \ldots E_n)\) corresponds to:

- OCAML’s \((E_1; \ldots ; E_n)\)
- SCHEME’s \((\text{begin } E_1 \ldots E_n)\)
- JAVA and C’s \(\{E_1; \ldots ; E_n;\}\) (no value returned)

What is the behavior of the following HOILEC expression?

\[(\text{bind } a (\text{cell } (+ 3 4)))
(\text{seq } (\text{println } (~ a)))
(\text{:= } a (* 2 (~ a)))
(\text{println } (~ a))
(\text{:= } a (+ 1 (~ a)))
(\text{println } (~ a))
(\text{bind } b (\text{cell } (~ a))
(\text{bind } c b
(\text{seq } (\text{println } (\text{cell=? } a b)))
(\text{println } (\text{cell=? } b c))
(\text{:= } c (/ (~ c) 5))
(\text{println } (~ a))
(\text{println } (~ b))
(~ c))))))\]

Unlike in HOFL, the order of evaluation of primitive operands makes a difference in HOILEC, and is specified to be left-to-right.\(^1\) For example, the following expressions can distinguish left-to-right and right-to-left evaluation of operands

\[(- (\text{println } (* 3 4)) (\text{println } (+ 1 2)))\]

\[(\text{bind } c (\text{cell } 1)
(+ (\text{seq } (\text{:= } c (* 10 (~ c))) (~ c))
(\text{seq } (\text{:= } c (+ 2 (~ c))) (~ c))))\)

\[(\text{bind } d (\text{cell } 1)
(+ (\text{:= } d 2) (* (\text{:= } d 3) (~ d))))\]

\(^1\)Even in HOFL, order of evaluation can be distinguished by error messages.
2 HOILEC Examples

2.1 Imperative Factorial

Here is an imperative factorial in JAVA:

```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 ;
}
```

Here is how we can express an imperative factorial in HOILEC:

```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))))
```

We can define the following while-loop syntactic sugar in HOILEC to express loops:

```hoilec
(while Etest Ebody)
  ;~
  (bindrec ((Iloop ; Iloop is fresh
               (fun ()
                 (if Etest
                    (seq Ebody (Iloop))
                    #f))) ; Arbitrary return value
             (Iloop) ; Start the loop
           )
```

For example:

```hoilec
(def (fact n)
  (bindpar ((num (cell n))
            (ans (cell 1)))
  (seq (while (> (^ num) 0)
        (seq (:= ans (* (^ num) (^ ans)))
             (:= num (- (^ num) 1)))
        (^ ans)))
```

We can modify this to print the state variables in the loop:
holoec> (def (fact n)
   (bindpar ((num (cell n))
             (ans (cell 1)))
   (seq (while (> (^ num) 0)
         (seq (print "(^ num) = ")
              (print (^ num))
              (print "; (^ ans) = ")
              (println (^ ans))
              (:= ans (* (^ num) (^ ans)))
              (:= num (- (^ num) 1))))
   (^ ans)))

fact

holoec> (fact 5)
"(^ num) = "5"; (^ ans) = "1
"(^ num) = "4"; (^ ans) = "5
"(^ num) = "3"; (^ ans) = "20
"(^ num) = "2"; (^ ans) = "60
"(^ num) = "1"; (^ ans) = "120
120
2.2 Collecting the Arguments to fib

Below is a HOILEC Fibonacci program that collects all the arguments to \texttt{fib} (in reverse order):

\begin{verbatim}
(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))))))
\end{verbatim}

For example:

\begin{verbatim}
# HoilecEnvInterp.runFile "fib-args.hec" [5];;
(list 5 (list 1 0 1 2 3 0 1 2 1 0 1 2 3 4 5))
\end{verbatim}

In HOFL, which does not have mutable cells, we would need to “thread” state through computation:

\begin{verbatim}
(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))))))
\end{verbatim}

2.3 Mutable Stacks in HOILEC

We can represent a mutable stack in HOILEC as a cell that contains a list of stack elements arranged from top down:

\begin{verbatim}
(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)))
\end{verbatim}

For example:

\begin{verbatim}
hoilec> (bind s (make-stack)
  (seq (push! 2 s) (push! 3 s) (push! 5 s)
    (+ (pop! s) (pop! s)))
  8
\end{verbatim}
2.4 fresh: 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)))))))
```

For example:

```
hoilec> (fresh "foo")
"foo.0"

hoilec> (fresh "bar")
"bar.1"

hoilec> (fresh "foo")
"foo.2"
```

Here is the implementation of StringUtils.fresh in OCAML:

```
(* fresh creates a "fresh" name for the given string
  by adding a "." followed by a unique number.
  If the given string already contains a dot,
  fresh just changes the number. E.g., fresh "foo.17"
  will give a string of the form "foo.XXX" *)

let fresh =
  let counter = ref 0 in
    fun str ->
      let base = (try let i = String.index str "." in String.sub str 0 i
        with Not_found -> str) in
        let count = !counter in
        let _ = counter := count + 1 in
        base ^ "." ^ (string_of_int count)
```

2.5 Promises in HOILEC

- (delayed EThunk) Return a promise to evaluate the thunk (nullary function) denoted by EThunk at a later time.

- (force EPromise) If the promise denoted by EPromise 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))))
  (delayed (fun () (println (inc!)))))
  (+ (force p) (force p)))))
```
Here is one way to implement promises in HOILEC:

```lisp
(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))))
```

Here is a second way to implement promises in HOILEC:

```lisp
(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))
```

## 2.6 Object-Oriented Stacks in HOILEC
3 Implementing the HOILEC Interpreter

4 Discussion

4.1 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 \texttt{set-car!} & \texttt{set-cdr!}) and vectors with mutable slots (modified via \texttt{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/output (I/O) operations for reading from/writing to files.

4.2 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 \texttt{StringUtils.fresh} to generate fresh names – avoids threading name generator throughout entire mini-language implementation.
  - Tracing functions in OCAML and SCHEME.
- Computational objects with local state are nice for modeling the real world. E.g., gas molecules, digital circuits, bank accounts.

4.3 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 $(\ast \ 2 \ E)$. But not true in the presence of side effects! E.g. $E = (\texttt{seq} \ (\texttt{:=} \ c \ (+ \ (^{\ c} \ 1)) ) \ a)$.

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

  $$(\texttt{if } E_1 \ E_2 \ E_3)$$
  $$<=?=> (\texttt{bind } I \ E_3 \ (\texttt{if } E_1 \ E_2 \ I)) \ ; \ I \ \texttt{fresh}$$

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

  $$((\ast \ (^{\ a})) \ (\texttt{seq} \ (\texttt{:=} \ b \ (\ast \ (+ \ (^{\ b}) \ (^{\ a})))) \ (^{\ a})))$$
  $$<=?=> (\texttt{seq} \ (\texttt{:=} \ b \ (\ast \ (+ \ (^{\ b}) \ (^{\ a})))) \ (\ast \ (2 \ (^{\ a}))))$$

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