CS320 Handout 12
Side Effects, Aliasing, Managing Share State

Typical Scheme programs are mostly written in a pure functional style. But imperative features have their place in programming, as many computational tasks are more conveniently implemented when it is possible to allocate (or mutate) storage during program evaluation. For example, this is the more natural way to do input and output. So, Scheme, like other functional languages, support imperative features. Programming in this style is commonly called programming with *side-effects*.

1 How Not To Program With Side-Effects

On the left is an implementation of the factorial function in Scheme, written in an imperative style. On the right is an implementation of the factorial in C.

```
;; Scheme implementation                              ;    C implementation
;; ---------------------                              ;    ----------------

   (define (imperative-fact n)                        ;    int fact(int n)
     (let ((count 0) (product 1))                     ;    {  int count = 0, product = 1;
        (define (loop)                                ;       while
           (if (< count n)                            ;         (count < n)
               (begin (set! count (+ count 1))        ;         {  count = count + 1;
                      (set! product (* product count)) ;            product = product * count;
                      (loop))  ))                     ;          };
        (loop)                                        ;       return product;
        product))                                     ;    }
```

The C implementation was written first, and the Scheme implementation was written second, so that it closely mimics the first. Perhaps the C implementation is as clear as we can make it in C. But the Scheme implementation is atrocious: Not only because it is needlessly complicated (although it is not much more complicated than the C implementation), but more importantly because there is no good reason to use side-effects in this case. Compare, for example, with the following Scheme implementation:

```
(define (fact n)
     (if (<= n 0) 1 (* (fact (- n 1)) n)))
```

The preceding is not only much simpler, but also follows closely the mathematical definition of the factorial function, which makes it easier to understand and verify.

For comparison purposes, here is the SML program that mimics the C implementation of the factorial function:

```
fun imperative_fact (n: int) =
    let val count   = ref 0
        val product = ref 1
        fun loop () =
            if (!count < n)
            then (count := !count + 1;
                  product := !product * !count;
                  loop () )
            else ()
    in
        loop (); !product
    end;
```

The SML code for `imperative_fact` is a little clearer than the Scheme code for `imperative-fact`, if only because the variables `count` and `product` are explicitly declared as *reference cells* from the very beginning of the program and retain their identity throughout. (Although there are subtle differences and sometimes ambiguities in the way people use these phrases, "reference cells" are also called "mutable cells", or "memory locations" by C programmers, or "state variables" by Scheme programmers.)

In the Scheme code, by contrast, `count` and `product` are initially names for integer values (`0` and `1` respectively), and they retain this identity to the end of the computation, if the input `n` is `0`. However, both `count` and `product` lose their initial identity and become *state variables* if the computation is started on an input `n` different from `0`.

Although the SML `imperative_fact` is arguably more transparent than the Scheme `imperative-fact`, there is again no reason to avoid a functional style of coding the factorial in SML:

```
fun fact (n : int) : int =
    if (n <= 0) then 1 else fact(n - 1) * n;
```

This functional definition is easy to read and verify and cannot be any simpler.

2 New syntax features

We take a closer look at the imperative features used in the Scheme translation of the C program for the factorial function, on page 1. (We leave for now the imperative features of the SML translation on page 2.)

Assignments

A `set!`-expression in Scheme is a special form, and it accomplishes the same thing as an assignment in other programming languages. It has the following shape:

`(set!` ⟨*variable*⟩ ⟨*expression*⟩`)`

The ⟨*variable*⟩ is the left-hand side, and ⟨*expression*⟩ is the right-hand side, of the assignment. The left-hand side being assignable, i.e., referring to a memory cell, is now called a *state variable*.

To use *state variables* in Scheme they first have to be declared, as all other variables. To update state variables, we use the special form `set!`, as well as other special forms, all with names ending with "!". Without prior declaration of a state variable, we get an error; for example, the following causes an error:

```
(set! x 5)                ;; x is not yet declared
```

But the following is correct:

```
(define x 0)              ;; x is declared and initialized to 0
(set! x 5)
```

Use side-effects with much care. Remember also that different Scheme interpreters deal differently with some expressions with side-effects.[1] Consider, for example, the following Scheme code:

```
(define x)                ;; x is declared but not initialized
(set! x 5)
```

MIT Scheme will not raise an error, but UMB Scheme and Guile Scheme will. Or consider:

```
(define y 0)
(define z (set! y 5))
z
```

This is legal code for MIT Scheme, UMB Scheme, Guile Scheme, and other Scheme interpreters. But what should be the value of `z`? Keep in mind that a `set!` expression is supposed to return no value: It is only used for its side-effect. Well, MIT Scheme binds `z` to the value of `y` *before* it is updated to 5:

```
1 ]=> (define y 0)
;Value: y
1 ]=> (define z (set! y 5))
;Value: z
1 ]=> z
;Value: 0
1 ]=>
```

Whereas UMB Scheme binds `z` to the value of `y` *after* it is updated to 5:

[1] This is so because the Scheme language specification – the most recent is called R5RS – does not specify what the values of these expressions are.

```
==> (define y 0)
y
==> (define z (set! y 5))
z
==> z
5
==>
```

And Guile Scheme binds `z` to nothing:

```
guile> (define y 0)
guile> (define z (set! y 5))
guile> z
guile>
```

Sequencing

A typical situation in imperative programming is to put in sequence several commands that affect the state, i.e., that have side-effects. This is the purpose of a `begin` expression: It makes it convenient to collect together several expressions whose only purpose is their side-effects. The shape of a `begin`-expression is:

`(begin` ⟨*expression-1*⟩
⟨*expression-2*⟩
...
⟨*expression-n*⟩
⟨*expression*⟩`)`

The evaluation determines the values of the $n+1$ subexpressions in order, ignores the first n values, and returns the value of the last subexpression as the value of the entire `begin`-expression. To be useful at all, the first n subexpressions have side-effects; only the last subexpression has an interesting value. Here is an example:

```
(define x 0)
(define y 2)
(define z 5)
(begin (set! x y)            ;  side-effect
       (set! y (+ y 2))      ;  side-effect
       (set! x 3)            ;  side-effect
       (list x y z))         ;  returns (3 4 5) for the entire begin-expression
```

Other new syntax features

In the Scheme code for `imperative-fact` on page 1, there are two features we have not used before:

- We use the `if` special form with only 1 branch instead of the usual 2 branches: If the test `(< count n)` is false, nothing happens (the else-branch of the "`if`" is blank).
- The loop in `imperative-fact` is implemented as a parameterless procedure.

Exercise: What happens if we replace all occurrences of "`(loop)`" in `imperative-fact` – there are 3 of them – by just "`loop`"?

3 Pitfalls of Side-Effects

A purely functional implementation of the factorial function, but now in tail-recursive form, is:

```
(define (fact n)
   (define (iter product count)
      (if (> count n)
          product
          (iter (* count product)
                (+ count 1))))
   (iter 1 1))
```

This is taken from the [A&S] text, in Section 1.2.1 and again in Section 3.1.3. The preceding can be translated into imperative style, turning `count` and `product` into state variables and using assignment (with `set!`) to update them:

```
(define (fact n)
   (let ((count 1)
         (product 1))
     (define (iter)
        (if (> count n)
          product
          (begin (set! product (* count product))  ;; A
                 (set! count (+ count 1))          ;; B
                 (iter))))
     (iter)))
```

Although correct, this translation into imperative style is not really called for. In contrast to the pure functional style, we now have to be careful about the order of the assignments: With side-effects, a temporal aspect is introduced and the order of evaluation matters. Switching the order of the assignments `A` and `B` above results in a buggy program:

```
(define (fact n)
   (let ((count 1)
         (product 1))
     (define (iter)
        (if (> count n)
          product
          (begin (set! count (+ count 1))          ;; B
                 (set! product (* count product))  ;; A
                 (iter))))
     (iter)))
```

The preceding gives you but a small taste of what we have to worry about with imperative programs: Their semantics are a lot more complicated than those of functional programs. And they become a major headache when concurrency is introduced (some of the issues are discussed in Section 3.4 of the [A&S] book).

4 Side-Effects That Are Called For

Many computational tasks are more naturally implemented in an imperative style. The material in this section is partly taken from pages 223 and 233 and in the [A&S] text. Consider the following Scheme code:

```
(define (make-account balance)                  ; BALANCE is a local state-variable

   (define (withdraw amount)                    ; local procedure WITHDRAW
      (if (>= balance amount)
          (begin
              (set! balance (- balance amount))
              balance)
          "Insufficient funds"))

   (define (deposit amount)                     ; local procedure DEPOSIT
      (set! balance (+ balance amount))
      balance)

   (define (dispatch m)                         ; local procedure DISPATCH
      (cond ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit)  deposit)
            (else (error "Unknown request -- MAKE-ACCOUNT" m))))

   dispatch)                                    ; MAKE-ACCOUNT is higher-order,
                                                ; it returns procedure DISPATCH
```

The procedure `make-account` can be used as follows:

```
1 ]=> (define acc1 (make-account 100))
;Value: acc1
1 ]=> ((acc1 'withdraw) 50)
;Value: 50
1 ]=> ((acc1 'withdraw) 60)
;Value 1: "Insufficient funds"
1 ]=> ((acc1 'deposit) 40)
;Value: 90
1 ]=> ((acc1 'withdraw) 60)
;Value: 30
```

There are several things to notice in relation to `make-account`: In addition to the use of a local state-variable, `balance`, to keep track of repeated `'withdraw` and `'deposit`, *message-passing* style is used.

Every time `make-account` is invoked, private (i.e., local) copies of the procedures `withdraw` and `deposit` are created. Thus, another call to `make-account`:

```
(define acc2 (make-account 100))
```

produces a totally separate account, `acc2`, which maintains its own local state-variable `balance`. Note that the following code:

```
(define acc1 (make-account 100))
(define acc2 (make-account 100))
```

is very different from the following:

```
(define acc1 (make-account 100))
(define acc2 acc1)
```

In the first case, there are 2 distinct accounts, so that transactions made by `acc1` will not affect `acc2`, and vice-versa. In the second case, `acc2` refers to the same thing as `acc1`, i.e., `acc1` and `acc2` are said to be *aliases* because they refer to the same thing.

5 Mutable List Structures

Section 3.3 in the [A&S] text gives several examples of compound data which are more conveniently represented by mutable list structures. In a pure functional style, compound data is created using *constructors* (e.g., the primitive `cons`) and dissected using *selectors* (e.g., the primitives `car` and `cdr`). With mutable list structures, we use not only constructors and selectors, but also *mutators*. The role of mutators is to modify and update compound data. There are primitive mutators `set-car!` and `set-cdr!`, and you can also define your own mutators according to need. The following examples are taken from the [A&S] text.

```
(define x (list (list 'a 'b) 'c 'd))          ;; [A&S] page 252
(define y (list 'e 'f))
(set-car! x y)
x                                             ;; ((e f) c d)

(define z (cons y (cdr x)))                   ;; [A&S] page 253
z                                             ;; ((e f) c d)

(define x (list (list 'a 'b) 'c 'd))          ;; [A&S] page 253
(define y (list 'e 'f))
(set-cdr! x y)
x                                             ;; ((a b) e f)

(define x (list 'a 'b))                       ;; [A&S] page 257
(define z1 (cons x x))
(define z2 (cons (list 'a 'b) (list 'a 'b)))

(define (set-to-wow! x)                       ;; [A&S] page 258
    (set-car! (car x) 'wow)
    x)

z1                                            ;; ((a b) a b)
(set-to-wow! z1)                              ;; ((wow b) wow b)

z2                                            ;; ((a b) a b)
(set-to-wow! z2)                              ;; ((wow b) a b)
```