CSci 2041: Advanced Programming Principles
Side Effects in Programming

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Lectures in Spring 2015

Side Effects: What and Why?

The primary model for communication up to now has been through expression evaluation

However, this model may not be natural in some situations

- communication may need to happen in an “unstructured” way; e.g. it may be between sibling expressions
- communication may be with an external process/system

  E.g. input from a user, input that is too large to provide directly, persistence of results
- there may be no natural way to go on with a computation

  E.g. a “divide-by-zero” situation, unanticipated form for data, user may provide bad input

We will understand constructs for treating such aspects and also how to use them systematically in programming

State as a Means for Unstructured Communication

Communication between sibling expressions can always be orchestrated through the parent

- the “producer” returns the value to the parent expression
- the parent passes the value to the “consumer”

The problem with this approach: the parent has to consider communication which is orthogonal to the original problem

The solution: let communication take place via an auxiliary “state” component

```
...    x := 5     ...

         ...
...    x + 3      ...
```

Treating State in Programming

We have already seen how to program state and related imperative constructs in a functional way

- we thread a state parameter through all the functions: it is part of the input as well as the output
- the producer of a value updates the state when returning it
- the consumer of a value looks it us as needed in the state

The benefit of this approach: the parent process only needs to worry about passing the state, not about how it is used

However, treating state explicitly can be costly; can we do better?

Modelling State Implicitly in OCaml

Four main components underlie the efficient treatment of state

- State is treated as an implicit collection of *typed references* to cells on the heap
- Constructors are provided for creating and for describing the types of such cells
  - the type constructor `ref` indicates the type of references; e.g. `(int ref)`
  - the value constructor `ref` creates such cells, e.g.

    ```
    # let x = ref 5;;
    val x : int ref = {contents = 5}
    #
    ```

    Note also the type associated with `x`

Modelling State Implicitly in OCaml (Contd)

- The infix operator `:= : ('a ref) -> 'a -> unit` is provided as a means for updating reference contents, e.g.

  ```
  # x := 7;;
  - : unit = ()
  #
  ```

  Note also the return type; there is no “meaningful” value for assignment, only a side-effect

- The prefix operator `! : 'a ref -> 'a` is provided as a means for looking up the value in a reference cell, e.g.

  ```
  # !x;;
  - : int = 7
  #
  ```

Sequencing of Expressions

When we have side effects, it can become important to sequence the evaluation of expressions

Two ways to do this

- Use nested *lets* that don't actually bind anything

  For example consider

```
# let _ = (x := 7)
    in let _ = y := !x + 2
          in !y;;
- : int = 9
#
```

Sequencing of Expressions (Contd)

- Use the expression sequencing construct

  In OCaml you can write `(e1; ...; en)`, which has the following effect:

  - expressions `e1`, ..., `en` are evaluated in sequence
  - the value and type of the expression is that of `en`

  For example, consider

```
# (x := 7; y := !x + 2; !y);;
- : int = 9
#
```

  Note: OCaml expects (but does not require) all expressions but the last to have type `unit`

State and the Order of Evaluation

The consumer must use a value only after it has been produced

With assignments, this means that we must pay careful attention to the order in which expressions are evaluated

For example, consider the following OCaml code

```
# let a = ref 1
# let b = ref 5
# let plus x y = x + y
# plus (a := 3; !a) (b:= !a + !b; !b);;
- : int = 9
#
```

Two consequences of this sensitivity to evaluation order

- knowledge of the order of evaluation must be explicit
- the “functional” elegance is compromised by side-effects

Other “State-Based” Constructs

With the notion of state builtin, OCaml provides control-flow contructs familiar from other languages, e.g.

- The *if-then-else* statement is a special case of the *if-then-else* expression

  Also, `if c then e1` abbreviates
  `if c then e1 else ()`

- There are `while` and `for` constructs in OCaml

  For example, consider

```
#   let x = ref 0
    in ( while (!x < 5) do x := !x + 1 done;
         !x );;
- : int = 5
#
```

Sermon: Resist the temptation to use such constructs, the functional elegance can be *really* useful for correctness!

Implicit versus Explicit Uses of Pointers

OCaml gives us the use of pointers even without references

For example, consider the following in C

```
typedef struct cell *intlist;
struct cell { int hd; intlist tl;}
```

This structure is mirrored in the OCaml type declaration

```
type intlist = Null | Cons' of cell
 and cell = {hd:int; tl : intlist;};;
```

The main difference: we do not deal with pointers explicitly, creating cells and “looking up” pointers happens implicitly

Note also that in OCaml such declarations can be polymorphic

Data with Mutable Fields

When pointers are used implicitly, we can create new values but not *update* old ones

To make components of data updateable, we can use references

```
type 'a mylist =
   Nil | Cons of ('a ref) * ('a mylist ref)
```

With this declaration, we can now destructively change the head and the tail of lists, e.g.

```
# let l = Cons(ref 1, ref Nil);;
# let Cons (h,t) = l in h := 2;;
# l;;
- : int mylist =
    Cons ({contents = 2}, {contents = Nil})
```

Mutable Data Structures

The data structures we have been using up to now have all been *immutable*

For example, our programs have had the following character:

- if they append lists, they may reuse old lists but will not change them
- if they insert in trees, they may reuse subtrees but will not change any existing ones

With the introduction of references, we might consider making them mutable

But is this a good choice? Not necessarily!

Mutable versus Immutable Data

Some issues to consider in deciding the tradeoff

- If you want predictable and modular behaviour, the issue is not *whether to copy* but, rather, *what to copy*

  For example, if you do change old lists, you would not want to reuse incoming ones but copy them

  E.g. in *(append l1 l2)* you would copy *l2* now

- If you do not use *deep copying* then the behaviour of programs with mutable data becomes difficult to predict

  The "functional elegance" really makes a difference in reasoning about programs!

- Mutable data structures are not necessary for efficiency; *functional data structures* can offer this too

  See *Purely Functional Data Structure* by Chris Okasaki

Outputting to an External Medium

Output to a file, to the terminal, etc, involves the following

- *Opening* a channel for the output using

  ```
    open_out : string -> out_channel
  ```

  This takes a file name and makes it a channel for printing

  Two predefined channels: `stdout` and `stderr`

- *Writing* to the channel using, e.g.

  ```
     Printf.fprintf: out_channel ->
       <fmt string> -> a1 -> ... -> an -> unit
  ```

  The `<fmt string>` describes how to print the arguments and the compatability is type-checked

  When the channel is `stdout`, we can use `printf`

- *Closing* the channel when finished using

  ```
  close_out : out_channel -> unit
  ```

Outputting to an External Medium (Example)

An interaction showing how to print text to a file and terminal

```
# Printf.printf "The value of %s is %d\n" "x" 5;;
The value of x is 5
- : unit = ()
# let outfile = open_out "myfile.txt";;
val outfile : out_channel = <abstr>
# Printf.fprintf outfile
                 "The value of %s is %d\" "x" 5;;
- : unit = ()
# close_out outfile;;
- : unit = ()
#
```

After this interaction, the file `myfile.txt` will contain

```
The value of x is 5
```

Look up Chapter 10 of Hickey's book, *Pervasives* in the *core library* and *Printf* in the *standard library* for more functions

Input from an External Medium

Analogous to output, reading from a file, terminal, etc, has the following components

- Opening an input channel using

  ```
  open_in : string -> in_channel
  ```

  This takes a file name and makes it a channel for reading

  Predefined channel for reading: `stdin`

- Reading from the channel using, e.g.

  ```
  input_char : in_channel -> char
  input_line : in_channel -> string
  ```

  For reading from `stdin` we can also use `read_line`

- Closing an input channel when finished using

  ```
  close_in : in_channel -> unit
  ```

Input from an External Medium (Example)

An interaction showing how to read text from a file

```
# let infile = open_in "myfile.txt";;
val infile : in_channel = <abstr>
# let s = input_line infile;;
val s : string = "The value of x is 5"
# close_in infile;;
- : unit = ()
#
```

Again, look up Chapter 10 of Hickey's book and *Pervasives* in the *core library* in the *standard library* for more functions

Also, for *formatted* reading, check out *Scanf* in the *standard library*

One issue with formatted input: you have to be wary of ill-formed content in files or in what the user presents

Input/Output and Expression Evaluation

Input/output is a computation for which timing matters

- it is something that impacts state or depends on state and so when it occurs makes a difference
- the order in which output is presented is also important for it to be useful to the external process

Thus, for i/o to be sensible, we must known enough to be able to predict the order of evaluation for relevant expressions

Dealing with Exceptional Situations

Three components are central to this capability

- being able to indicate the kind of situation encountered

  Done through *exception declarations* of the form

  ```
  exception  <Name> [of <type>]
  ```

- signalling that the situation has been encountered

  Done by *raising* an exception via an expression of the form

  ```
  raise (<Name> (<arg1>, ..., <argn>))
  ```

- anticipating such situations and preparing to handle them

  Done by setting up *exception handlers* via expressions of the form

  ```
   try <expression> with
    | <Exc-Name> (<p11>, ..., <p1n1>) -> <expression>
          ...
    | <Exc-Name> (<p1m>, ..., <p1nm>) -> <expression>
  ```

Using Exceptions (Example)

A example illustrating the different aspects of exceptions

```
exception NilList
exception Singleton of int

let head_aux l =
  match l with
  | [] -> raise NilList
  | [x] -> raise (Singleton x)
  | (h::t) -> h

let head l =
  try Printf.printf
         "Head element %d\n" (head_aux l) with
  | NilList -> Printf.printf "Empty list\n"
  | Singleton x ->
        Printf.printf "Sole element %d\n" x
```

Some Odds and Ends of Exceptions

- Uncaught exceptions trickle up the calling chain till a handler is found or till the top-level is reached

```
let head l =
  try Printf.printf
         "Head element %d\n" (head_aux l) with
  | NilList -> Printf.printf "Empty list\n"

# try (head [1]) with
  | Singleton x ->
        Printf.printf "Sole element %d\n" x
Sole element 1
- : unit = ()
#
```

- The *option* type can sometimes be used instead of exceptions but each recursive call has to have a“handler”
- For more on exceptions, read Chapter 9 of Hickey’s book