CS 2SC3 and SE 2S03 Fall 2009

03 Imperative Programming

William M. Farmer

Department of Computing and Software
McMaster University

19 October 2009

Data Structures

- A data structure is a portion of memory that holds a structured collection of values.
  - A data structure may be itself a value.
- Various operators are associated with each kind of data structure:
  - Constructors for creating data structures.
  - Selectors for retrieving the values in data structures.
  - Mutators for modifying the values in data structures.
- Access to these operators needs to be controlled to ensure data privacy, integrity, and availability.

Mutable vs. Immutable Data Structures

- A data structure is mutable [immutable] if it has [does not have] mutators.
  - Many data structures are immutable.
- The imperative programming paradigm heavily uses mutable data structures.
- The functional programming paradigm avoids using mutable data structures.

Example: Pairs

- A pair is a data structure that holds an ordered pair $\langle a, b \rangle$ of two values a and b with unspecified types.
- Constructor:
  - pair(a, b) creates a data structure p holding $\langle a, b \rangle$.
- Selectors:
  - get-fst(p) returns a, the first value in p.
  - get-snd(p) returns b, the second value in p
- Mutators:
  - set-fst(p, x) sets a, the first value in p, to x.
  - set-snd(p, x) sets b, the second value in p, to x.
  - Note: a and x (as well as b and x) need not have the same type.

Use of Pairs

- The pair data structure can be used to build many other useful data structures.
  - It is the chief data structure of Lisp.
- Pairs can be used to define tuples:

  $(a_1, a_2) = \langle a_1, a_2 \rangle$.
  $(a_1, \ldots, a_n) = \langle a_1, (a_2, \ldots, a_n) \rangle$ for $n \geq 3$.
- Pairs can be used to define lists:
  - $[\,] =$ nil, some special value.
  - $[a_1] = \langle a_1, [\,] \rangle$.
  - $[a_1, \ldots, a_n] = \langle a_1, [a_2, \ldots, a_n] \rangle$ for $n \geq 2$.

Example: References

- A reference of type t is a data structure that holds a value of type t.
- A reference is said to reference or point to its value.
- In OCaml, `ref` is a polymorphic type of references.
- Constructor: `ref` expr constructs a reference of the type of t `ref` where t is the type of expr.
  - Example: `let x = ref 8 ;;`
- Selector: If expr is a reference, `!`expr selects the referenced value of the reference.
  - Example: `!x ;;`
- Mutator: If expr_1 is a reference of type t and expr_2 is a value of type t, then expr_1 `:=` expr_2 sets the referenced value of expr_1 to expr_2.
  - Example: `x := 7 ;;`

Control Structures

- A control structure controls the execution of statements in a program.
- Before control structures were invented, execution was controlled in an unstructured manner using conditionals and goto statements.
  - This made the control flow of the program exceedingly difficult to understand.

Kinds of Control Structures

- There are three main categories of control structures:
  1. Sequential control structures allow a sequence of statements to be executed one after another.
  2. Conditional control structures allow a statement to be selected for execution on the basis of whether a condition evaluates to true or false.
  3. Iterative control structures allow a statement to be repeatedly executed.
- There are several kinds of control structures in each of these categories.
- An imperative programming language must have at least one control structure from each of these three categories in order to be Turing complete.

Block

- A block is a sequential control structure that treats a sequence of statements as a single statement.
- The statements in a block are executed left to right.
- OCaml has two syntaxes for blocks:

  `(`expr_1`;` $\cdots$ `;` expr_n`)`

  `begin` expr_1 `;` $\cdots$ `;` expr_n `end`

If-Then-Else Statements

- An if-then-else statement is a conditional control structure in which one of two statements is selected on the basis of whether a condition evaluates to true or false.
- OCaml has the following syntax for if-then-else statements:

  `if` expr_1 `then` expr_2 `else` expr_3

  where expr_1 is of type `bool` and expr_2 and expr_3 are of type `unit`.
- An OCaml if-then-else statement is a special case of an OCaml conditional expression.
- The form

  `if` expr_1 `then` expr_2

  is equivalent to

  `if` expr_1 `then` expr_2 `else ()`

For Loop

- The for loop is an iterative control structure that executes a statement for a certain number of times.
- For loop iteration is normally bounded.
- OCaml has two syntaxes for for loops:

  `for` name `=` expr_1 `to` expr_2 `do` expr_3 `done`

  `for` name `=` expr_1 `downto` expr_2 `do` expr_3 `done`

  where expr_1 and expr_2 are expressions of type `int` and expr_3 is an expression of type `unit`.

While Loop

- The while loop is an iterative control structure that executes a statement as long as a condition is true.
- While loop iteration is unbounded.
- The while loop is more general than the for loop; it can simulate a for loop.
- OCaml has the following syntax for while loops:

  `while` expr_1 `do` expr_2 `done`

  where expr_1 is of type `bool` and expr_2 is of type `unit`.

Loop Termination

- The execution of a loop may never terminate.
  - This usually results in program failure!
- A loop terminates if there is a natural number value that strictly decreases with each iteration of the loop.
  - Loop termination is thus proved by showing that some natural number value strictly decreases with each iteration.
- Loop termination should be proved and documented for every loop in a program.

Loop Invariants

- An invariant of a loop is a expression E such that:
  1. E is true before the loop is executed for the first time.
  2. E is true after each execution of the body of the loop.
- An invariant can serve as a specification of the loop.
  - An invariant may be completely trivial.
- It is good practice to formulate the invariant of a loop before writing the loop.
- The documentation of a loop should include an invariant.
- Note: For some kinds of loop, such as the OCaml for loop, a loop invariant only makes sense when the loop is viewed as a while loop.

Example: A MinMax Program

```
let minmax x =
  let l = ref x in
  let min = ref infinity in
  let max = ref neg_infinity in
  for i = 0 to (List.length x) - 1 do
    let f = List.hd !l in
    if f < !min then min := f ;
    if f > !max then max := f ;
    l := List.tl !l
  done ;
  (!min,!max) ;;
```

Example: MinMax Correctness

- Claim 1: The following is a strictly decreasing natural number value for the loop:

  `(List.length x) - 1 - i`.

- Claim 2: The following is an invariant of the loop:

  $$\forall\, m : \texttt{int} \,.\, 0 \leq m \leq \texttt{i} \Rightarrow \texttt{!min} \leq \texttt{x}_m \leq \texttt{!max}$$

  where $\texttt{x}_m$ is the m-th component of `x`.

- Claim 3: Given a list x as input, `minmax` returns the minimum and maximum components of x.

  Proof. Following immediately from Claims 1 and 2.

Anomalies and Exceptions

- An anomaly is an unexpected behavior by a service.
- An exception is a signal to the client of a service that an anomaly has been exhibited by the service.
  - An exception is thrown or raised when the anomaly occurs.
- A thrown exception is caught by an appropriate exception handler that tries to handle the exception.
- Ways an exception can be handled:
  1. An attempt is made to recover from the anomaly.
  2. The exception is thrown higher up the uses chain.
  3. The state of the program providing the service is repaired as best as possible and then the service is allowed to fail.
- Exceptions can be used to change the normal flow of control.

Exceptions in OCaml

- Exception declaration:

  `exception` Name `;;`

- Raising an exception:

  `raise` Name `;;`

- Catching and handling an exception:

  `try` expr `with`
  `|` p_1 `->` expr_1
  ⋮
  `|` p_n `->` expr_n

Euclid's GCD Algorithm: Problem

- The greatest common divisor (GCD) of two positive integers is the largest positive integer that divides both integers without a remainder.
- Problem: Implement an OCaml function `gcd` of type

  ```
  int -> int -> int
  ```

  that computes the GCD of two positive integers.
- Some mathematical facts:
  1. If $x > 0$, $y > 0$, and $x > y$, then

     $$\text{GCD}(x - y, y) = \text{GCD}(x, y).$$

  2. If $x > 0$, $\text{GCD}(x, x) = x$.
- An algorithm that computes the GCD was described by Euclid in his Elements (c. 300 BC).

Euclid's GCD Algorithm: Solution

```
exception GCD_nonpositive_arguments ;;

let gcd x y =
  if x <= 0 || y <= 0
  then raise GCD_nonpositive_arguments
  else
    let xref = ref x in
    let yref = ref y in
    while !xref <> !yref do
      if !xref > !yref then xref := !xref - !yref
      else yref := !yref - !xref
    done ;
    !xref ;;
```

Euclid's GCD Algorithm: Correctness

- Claim 1: max(`!xref`, `!yref`) is a strictly decreasing natural number value for the loop.
- Claim 2: GCD(`!xref`, `!yref`) is an invariant of the loop.
- Claim 3: GCD(`x`, `y`) = `gcd x y`.

  Proof. By Claim 1, eventually `!xref` = `!yref`. Then

  $$\text{GCD}(\texttt{!xref}, \texttt{!yref}) = \texttt{!xref} = \texttt{gcd x y}.$$

  By Claim 2,

  $$\text{GCD}(\texttt{x}, \texttt{y}) = \text{GCD}(\texttt{!xref}, \texttt{!yref}).$$

  Therefore,

  $$\text{GCD}(\texttt{x}, \texttt{y}) = \texttt{gcd x y}.$$