OCaml Imperative Programming

CPSC 310 - Programming Languages

Imperative OCaml

- Functional programming supports mathematical reasoning because variables are immutable
- But, it is sometimes useful for values to change
  - for example, implementing an efficient hash table
- OCaml has references, fields, and arrays that are mutable

References

- `'a ref`: pointer to a mutable value of type `'a`
- There are three basic operations on references:
  - `ref : 'a -> 'a ref`: allocate a reference
  - ‘! : ’a ref -> ’a: read the stored value
  - `:= : 'a ref -> 'a -> unit`: change the stored value
- Binding a variable to a reference is immutable, but the contents of the reference is mutable

Reference Example

```
# let z = 3;;
val z : int = 3
# let x = ref z;;
val x : int ref = {contents = 3}
# let y = x;;
val y : int ref = {contents = 3}
# x := 4;;
- : unit = ()
# x;;
- : int ref = {contents = 4}
# !y;;
- : int = 4
```

Aliasing

- In the previous example

```
let z = 3;;
let x = ref z;;
let y = x;;
x := 4;;
!y;;
```

- The variables `x` and `y` are aliases:
  - in `let y = x` variable `x` evaluates to a location, and `y` is bound to the same location
  - so, changing the contents of that location causes both `!x` and `!y` to change

References: Syntax and Semantics

- Syntax: `ref e`
- Evaluation
  - evaluate `e` to a value `v`
  - allocate a new location `loc` in memory to hold `v`
  - store `v` in contents of memory at `loc`
  - return `loc` (which is a value)
- Type checking: `(ref e) : t ref` if `e : t`

References: Syntax and Semantics

- Syntax: `e1 := e2`
- Evaluation
  - evaluate `e2` to a value `v2`
  - evaluate `e1` to a location `loc`
  - store `v2` in contents of memory at `loc`
  - return `()`
- Type checking: `(e1 := e2) : unit` if `e1 : t ref` and `e2 : t`

References: Syntax and Semantics

- Syntax: `!e1`
- Evaluation
  - evaluate `e` to a location `loc`
  - return contents `v` of memory at `loc`
- Type checking: `!e : t` if `e : t ref`

References: Syntax and Semantics

- Syntax: `e1; e2`
  - same as 'let () = e1 in e2
- Evaluation
  - evaluate `e1` to a value `v1`
  - evaluate `e1` to a value `v2`
  - return `v2` (useful if `e1` has side effects since the value `v1` is thrown away)
- Type checking: `(e1; e2) : t` if `e1 : unit` and `e2 : t`

Grouping Sequences

- Parentheses or `begin ... end` can be used to group statements together; you may need this depending on the scoping rules.
- Example

```
let x = ref 0
let f () =
  begin  (* equivalent to "(" *)
    print_string "hello";
    x := !x + 1
  end    (* equivalent to ")" *)
```

Example: Implement a Counter with a Closure

```
# let next =
    let counter = ref 0 in
    fun () ->
      counter := !counter + 1; !counter;;
val next : unit -> int = <fun>
# next ();;
- : int = 1
# next ();;
- : int = 2
```

Order of Evaluation

- Consider this example

```
let y = ref 1
let f _ z = z + 1
let w = f (y := 2) !y
w
```

- If the order of evaluation is left-to-right, then `w` is 3
- If the order of evaluation is right-to-left, then `w` is 2

OCaml Order of Evaluation

- In OCaml, the order of evaluation is unspecified
  - So, different implementations can make different decisions
- Try to make your programs produce the same answer regardless of evaluation order

Structural vs. Physical Equality

- The = operator compares objects structurally
  - The <> operator is the negation of structural equality
- The == operator compares objects physically
  - The `!=` operator is the negation of physical equality
- Examples
  - `([1;2;3] = [1;2;3]) = true`
  - `([1;2;3] <> [1;2;3]) = false`
  - `([1;2;3] == [1;2;3]) = false`
  - `([1;2;3] != [1;2;3]) = true`
- The = operator is a problem with cyclical data structures

Cyclical Data Structure Example

- Type with operations:

```
type 'a rlist = Nil
  | Cons of 'a * ('a rlist ref)
let new_cell x y = Cons(x, ref y)
let update_next (Cons (_,r)) y = r := y
```

- Problem

```
let x = new_cell 1 Nil;;
update_next x x;;
x == x;;
x = x;; (* infinite loop *)
```

Equality of `refs`

- Refs are compared structurally by their contents, physically by their address
- Example

```
ref1 = ref1                  (* true *)
ref1 <> ref2                 (* true *)
ref1 != ref1                 (* false *)
let x = ref 1 in x == x      (* true *)
```

Mutable Record Fields

- Fields of a record type can be declared as mutable
- Example

```
# type point = {x:int; y:int; mutable c:string};;
type point = { x : int; y : int; mutable c : string; }
# let p = {x=0; y=0; c="red"};;
val p : point = {x = 0; y = 0; c = "red"}
# p.c <- "black";;
- : unit = ()
# p;;
- : point = {x = 0; y = 0; c = "black"}
# p.x <- 2;;
Error: The record field x is not mutable
```

Ref Implementation

- Ref cells are syntactic sugar

```
type 'a ref = { mutable contents: 'a }
let ref x = { contents = x }
let (!) r = r.contents
let (:=) r val = r.contents <- val
```

- The `ref` type is declared in the `Pervasives` module
- `ref` functions are compiled to equivalents above

Arrays

- Arrays generalize ref cells from a single mutable value to a sequence of mutable values
- Example

```
# let v = [|1; 2; 3|];;
val v : int array = [|1; 2; 3|]

# v.(0) <- 42;;
- : unit = ()
# v;;

- : int array = [|42; 2; 3|]
```

Arrays

- Syntax: `[|e1; ...; en|]`
- Evaluation
  - Evaluate to an n-element array with values are initialized to `v1`, `... vn` where `e1` evaluates to `v1`, `...`, `en` evaluates to `vn`
  - The evaluation order is right-to-left
- Type checking: `[|e1; ...; en|] : t array`
  - if for all `i` each `ei : t`

Arrays

- Syntax: `e1.(e2)`
- Evaluation
  - Evaluate `e2` to integer value `v2`
  - Evaluate `e1` to array value `v1`
  - If `0 <= v2 < n` where `n` is the length of array `v1` then return element at offset `v2` of `v1`
  - Otherwise, rais `Invalid_argument` exception
- Type checking: `e1.(e2) : t`
  - if `e1: t array` and `e2: int`

Arrays

- Syntax: `e1.(e2) <- e3`
- Evaluation
  - Evaluate `e3` to `v3`
  - Evaluate `e2` to integer value `v2`
  - Evaluate `e1` to array value `v1`
  - If `0 <= v2 < n` where `n` is the length of array `v1` then update the element at offset `v2` of `v1` to `v3`
  - Otherwise, rais `Invalid_argument` exception
  - Return `()`
- Type checking: `e1.(e2) <- e3 : unit`
  - if `e1: t array`, `e2: int` and `e3 : t`

Control Structures

- OCaml has loop structures that are useful when writing imperative code

```
while e1 do e2 done
for x=e1 to e2 do e3 done
for x=e1 downto e2 do e3 done
```