COS320: Compiling Techniques

Zak Kincaid

February 7, 2019
Today: OCaml cont’d
Review session

Today 6-8pm, room TBD
OCaml is an expression-oriented language

• An expression is something that evaluates to a value
  • Contrast to a statement, which expresses an action

• Example: In OCaml, variables are immutable
  • There is no statement can be used to over-write the value of a variable
OCaml is an expression-oriented language

- An expression is something that evaluates to a value
  - Contrast to a statement, which expresses an action
- Example: In OCaml, variables are immutable
  - There is no statement can be used to over-write the value of a variable
- Example: conditionals
  - In Java: if is a statement
    ```java
    if (x < 0) { x = -x; }
    ```
  - In OCaml: if is an expression
    ```ocaml
    if (x < 0) then -x else x
    ```
This is a matter of taste:

- OCaml has *reference cells*
  - let x = ref 0 in exp(ref ~ malloc in C)
  - Can over-write contents of reference cells: x := e
  - Can over-write fields of mutable records (~ C structs): rec.field <- e
  - Can over-write arrays: array.(i) <- e
This is a matter of taste:

- OCaml has **reference cells**
  - let \( x = \text{ref } 0 \) in \( \text{exp (ref } \sim \text{malloc in C) } \)
  - Can over-write contents of reference cells: \( x := e \)
  - Can over-write fields of mutable records (~ C structs): rec.field <- e
  - Can over-write arrays: array.(i) <- e

- OCaml has statements: ref cell assignment, for and while loops, sequencing
  - statements are expressions, which evaluate to () “unit”

```ocaml
let x = ref exp in (if (!x < 0) then x := -(!x) else (); !x)
```
This is a matter of taste:

- OCaml has *reference cells*
  - let x = ref 0 in exp (ref ~ malloc in C)
  - Can over-write contents of reference cells: x := e
  - Can over-write fields of mutable records (~~ C structs): rec.field <- e
  - Can over-write arrays: array.(i) <- e

- OCaml has statements: ref cell assignment, for and while loops, sequencing
  - statements are expressions, which evaluate to () “unit”

```ocaml
let x = ref exp in (if (!x < 0) then x := -(!x) else (); !x)
```

*Use sparingly*
Imperative BST

definition of a node

let insert key value tree =
    let current = ref tree in
    let continue = ref true in
    while !continue do
        match !(!current) with
        | Leaf ->
            (!current) := Node (key, ref value, ref Leaf, ref Leaf)
        | Node (k, v, left, right) ->
            if k = key then begin
                v := value;
                continue := false;
            end else if k < key then
                current := left
            else
                current := right
        done
type 'a tree =
  | Node of (int * 'a * 'a tree * 'a tree)
  | Leaf
let rec insert key value tree =
  match tree with
  | Leaf -> Node (key, value, Leaf, Leaf)
  | Node (k, v, left, right) ->
    if k = key then
      Node (k, value, left, right)
    else if k < key then
      Node (k, v, insert key value left, right)
    else
      Node (k, v, left, insert key value right)
Functions

• \((\text{fun } v \rightarrow e)\) is an expression, which evaluates to a value (closure)

• let \(f \ x \ y \ z = e\) is syntactic sugar for let \(f = \text{fun } x \rightarrow (\text{fun } y \rightarrow (\text{fun } z \rightarrow e))\)

• E.g., the type of \(*\) is not int * int -> int, it’s int -> (int -> int)

---

let rec iterate =
  fun (f:int -> int) ->
    fun (n:int) ->
      if n = 0 then
        (fun (x:int) -> x)
      else
        (fun (x:int) -> f (iterate f (n-1) x))

let exp base n = iterate (( * ) base) n 1
let two_to_five = exp 2 5
Algebraic data types

Simplest use-case: C-style enums

```haskell
type color = Red | Green | Blue

(* This type definition defines three constructors (Red, Green, and Blue), which evaluate to values of type color *)

let mycolor : color = Green

(* Can deconstruct using pattern matching (~ switch in C) *)

let to_string (c : color) =
  match c with
  | Red  -> "red"
  | Green -> "green"
  | Blue  -> "blue"
```
Unlike enums, each variant may contain a payload:

```ml
type point = float * float

type shape =
  | Rectangle of point * point
  | Circle of point * float
```

- Can be parameterized:
  ```ml
  type 'a option = None | Some of 'a
  ```

- Can be recursive:
  ```ml
  type expr = Var of string | Add of expr * expr | Mul of expr * expr
  ```

- Can be both:
  ```ml
  type 'a list = Nil | Cons ('a * 'a list)
  ```
Pattern matching binds variables to payload

```ocaml
type point = float * float
type shape =
  | Rectangle of point * point
  | Circle of point * float

let area (s:shape) =
  match s with
  | Rectangle (topleft, bottomright) ->
    (match topleft with
     | (tlx,tly) -> match bottomright with
      | (brx,bry) -> (brx -. tlx) *. (tly -. bry))
  | Circle (center, radius) -> pi *. radius *. radius
```
Pattern matching binds variables to payload

```ocaml
type point = float * float

let area (s: shape) =
    match s with
    | Rectangle (topleft, bottomright) ->
        match topleft with
        | (tlx, tly) ->
            match bottomright with
            | (brx, bry) -> (brx -. tlx) *. (tly -. bry)
    | Circle (center, radius) -> pi *. radius *. radius
```

Ambiguous!
Patterns can be nested

type point = float * float
type shape =
  | Rectangle of point * point
  | Circle of point * float

let area (s:shape) =
  match s with
  | Rectangle ((tlx,tly), (brx,bry)) -> (brx -. tlx) *. (tly -. bry))
  | Circle (_, radius) -> pi *. radius *. radius
A module groups together a collection of types and values

```ocaml
module IntSet = struct
  type elt = int
  type t = Leaf | Node of int * t * t
  let empty = Leaf
  let rec insert (e:elt) (s:t) = ...
end

module StringSet = struct
  type elt = string
  type t = Leaf | Node of string * t * t
  let empty = Leaf
  let rec insert (e:elt) (s:t) = ...
end

(* IntSet.empty != StringSet.empty *)
```
A module groups together a collection of types and values

```ocaml
module IntSet = struct
  type elt = int
  type t = Leaf | Node of int * t * t
  let empty = Leaf
  let rec insert (e:elt) (s:t) = ...
end

module StringSet = struct
  type elt = string
  type t = Leaf | Node of string * t * t
  let empty = Leaf
  let rec insert (e:elt) (s:t) = ...
end

(* IntSet.empty != StringSet.empty *)
```

- Each filename.ml file defines a module Filename
- Each filename.mli file defines the interface of Filename
- Some useful modules in the standard library: Int32, Int64, List, Printf, Format
A **functor** is a module that is parameterized by another module.

- **Set.Make**
  - **Input**: OrderedType module Ord, containing a type `t` and a function `compare` for comparing them
  - **Output**: Data structure representing sets of Ord.t’s

- **Map.Make**
  - **Input**: OrderedType module Ord, containing a type `t` and a function `compare` for comparing them
  - **Output**: Data structure representing maps with Ord.t keys