CMSC 330: Organization of Programming Languages

OCaml Data Types
OCaml Data

- So far, we’ve seen the following kinds of data
  - Basic types (int, float, char, string)
  - Lists
    - One kind of data structure
    - A list is either [ ] or h::t, deconstructed with pattern matching
  - Tuples and Records
    - Let you collect data together in fixed-size pieces
  - Functions

- How can we build other data structures?
  - Building everything from lists and tuples is awkward
User Defined Types

- `type` can be used to create new names for types
  - Useful for combinations of lists and tuples

- **Examples**
  - `type my_type = int * (int list)`
  - `let (x:my_type) = (3, [1; 2])`

  - `type my_type2 = int*char*(int*float)`
  - `let (y:my_type2) = (3, ‘a’, (5, 3.0))`
(User-Defined) Variants

type coin = Heads | Tails

let flip x =
    match x with
    Heads -> Tails
    | Tails -> Heads

let rec count_heads x =
    match x with
    [] -> 0
    | (Heads::x') -> 1 + count_heads x'
    | (_::x') -> count_heads x'

In simplest form:
Like a C enum

Basic pattern
matching
resembles C
switch

Combined list
and variant
patterns possible
Constructing and Destructing Variants

- **Syntax**
  - type $t = C_1 \mid \ldots \mid C_n$
  - the $C_i$ are called *constructors*
    - Must begin with a capital letter

- **Evaluation**
  - A constructor $C_i$ is already a value
  - Destructing a value $v$ of type $t$ is done by pattern matching on $v$; the patterns are the constructors $C_i$

- **Type Checking**
  - $C_i : t$ (for each $C_i$ in $t$’s definition)
Data Types: Variants with Data

- We can define variants that “carry data” too
  - Not just a constructor, but a constructor *plus values*

```
type shape =
  Rect of float * float (* width*length *)
| Circle of float (* radius *)
```

- **Rect** and **Circle** are constructors
  - where a `shape` is either a `Rect(w, l)`
    - for any floats `w` and `l`
  - or a `Circle r`
    - for any float `r`
Data Types (cont.)

- Use pattern matching to **deconstruct** values
  - Can bind pattern values to data parts
- Data types are *aka* algebraic data types and tagged unions

```ocaml
let area s =
  match s with
  Rect (w, l) -> w *. l
  | Circle r -> r *. r *. 3.14

area (Rect (3.0, 4.0));; (* 12.0 *)
area (Circle 3.0);; (* 28.26 *)
```
Data Types (cont.)

```ocaml
type shape =
  Rect of float * float (* width*length *)
| Circle of float       (* radius *)

let lst = [Rect (3.0, 4.0) ; Circle 3.0]
```

• What's the type of `lst`?
  • `shape list`

• What's the type of `lst`'s first element?
  • `shape`
public interface Shape {
    public double area();
}

class Rect implements Shape {
    private double width, length;

    Rect (double w, double l) {
        this.width = w;
        this.length = l;
    }

    double area() {
        return width * length;
    }
}

class Circle implements Shape {
    private double rad;

    Circle (double r) {
        this.rad = r;
    }

    double area() {
        return rad * rad * 3.14159;
    }
}
Option Type

```ocaml
type optional_int =
  None
| Some of int

let divide x y =
  if y != 0 then Some (x/y)
  else None

let string_of_opt o =
  match o with
    Some i -> string_of_int i
  | None -> "nothing"
```

- Comparing to Java: `None` is like `null`, while `Some i` is like an `Integer(i)` object
Polymorphic Option Type

- A Polymorphic version of option type can work with any kind of data
  - As int option, char option, etc...

\[
\text{type } 'a \text{ option } = \\
\quad \text{Some of } 'a \\
\mid \text{ None}
\]

In fact, this option type is built into OCaml

\[
\begin{align*}
\text{let } \text{opthd} l &= \text{match } l \text{ with} \\
& [ ] \rightarrow \text{None} \\
& x::_ \rightarrow \text{Some } x
\end{align*}
\]

\[
\begin{align*}
\text{let } p &= \text{opthd } [ ];; & (* p = \text{None } *) \\
\text{let } q &= \text{opthd } [1;2];; & (* q = \text{Some } 1 *) \\
\text{let } r &= \text{opthd } ["a"];; & (* r = \text{Some } "a" *)
\end{align*}
\]
type foo = (int * (string list)) list

Which one of the following could match foo?

A. [(3, "foo", "bar")]
B. [(7, ["foo"; "bar"])]
C. [(5, ["foo", "bar"])]
D. [(9, [(["foo", "bar"])]]}
type foo = (int * (string list)) list

Which one of the following could match foo?

A. [(3, “foo”, “bar”)]
B. [(7, [“foo”; “bar”])]  
C. [(5, [“foo”, “bar”])]  
D. [(9, [(“foo”, “bar”)])]
Quiz 2: What does this evaluate to?

```ocaml
type num = Int of int | Float of float;;
let plus a b =
    match a, b with
    | Int i, Int j -> Int (i+j)
    | Float i, Float j -> Float (i +. j)
    | Float i, Int j -> Float (i +. float_of_int j)
    ;;
plus (Float 3.0) (Int 2);;
```

A. **Float 5.0**
B. **5.0**
C. **Int 5**
D. **Type Error**
Quiz 2: What does this evaluate to?

type num = Int of int | Float of float;;

let plus a b =
    match a, b with
    | Int i, Int j -> Int (i+j)
    | Float i, Float j -> Float (i +. j)
    | Float i, Int j -> Float (i +. float_of_int j)

plus (Float 3.0) (Int 2);;

A. **Float 5.0**
B. 5.0
C. **Int 5**
D. Type Error
Quiz 3: What does this evaluate to?

```ocaml
let foo f = match f with
  None -> 42.0
  | Some n -> n +. 42.0

foo 3.3;;
```

A. 45.3  
B. 42.0  
C. Some 45.3  
D. Error
Quiz 3: What does this evaluate to?

let foo f = match f with
        None -> 42.0
    | Some n -> n +. 42.0

foo 3.3;;  foo (Some 3.3)

A. 45.3
B. 42.0
C. Some 45.3
D. Error
Recursive Data Types

- We can build up lists with recursive variant types

```
type 'a mylist =
    Nil
  | Cons of 'a * 'a mylist

let rec len = function
    Nil -> 0
  | Cons (_, t) -> 1 + (len t)

len (Cons (10, Cons (20, Cons (30, Nil))))
(* evaluates to 3 *)
```

- Won’t have nice \([1; 2; 3]\) syntax for this kind of list
Variants (full definition)

- **Syntax**
  - `type t = C1 [of t1] | ... | Cn [of tn]`
  - the $C_i$ are called constructors
    - Must begin with a capital letter; may include associated data notated with brackets $[]$ to indicate it’s optional

- **Evaluation**
  - A constructor $C_i$ is a value if it has no assoc. data
    - $C_i \, v_i$ is a value if it does
  - Destructuring a value of type $t$ is by pattern matching
    - patterns are constructors $C_i$ with data components, if any

- **Type Checking**
  - $C_i [v_i] : t$ [if $v_i$ has type $t_i$]
OCaml Exceptions

```ocaml
exception My_exception of int
let f n =
  if n > 0 then
    raise (My_exception n)
  else
    raise (Failure "foo")
let bar n =
  try
    f n
  with My_exception n ->
    Printf.printf "Caught %d\n" n
  | Failure s ->
    Printf.printf "Caught %s\n" s
```
Exceptions (cont.)

- Exceptions are declared with `exception`
  - They may appear in the signature as well
- Exceptions may take arguments
  - Just like type constructors
  - May also have no arguments
- Catch exceptions with `try...with...`
  - Pattern-matching can be used in `with`
  - If an exception is uncaught
    - Current function exits immediately
    - Control transfers up the call chain
    - Until the exception is caught, or until it reaches the top level
OCaml Exceptions (cont.)

- `failwith`: Raise exception `Failure` with the given string.
- `invalid_arg`: Raise exception `Invalid_argument` with the given string.
- `Not_found`: Raised if the object does not exist.

```ocaml
define div x y =
    if y = 0 then failwith "divide by zero" else x/y;;
define lst =[(1,"alice");(2,"bob");(3,"cat")];;
define lookup key lst =
    try
        List.assoc key lst
    with
        Not_found -> "key does not exist"
```