OCaml Data Types

CPSC 310 - Programming Languages

OCaml Data

- We have seen the following kinds of data
  - Basic types (int, float, char, string)
  - Lists
    - A list is either `[]` or `h::t`, deconstructed with pattern matching
  - Tuples and Records
    - Collect data in fixed size pieces
  - Functions
- Can we build other data structures?

User Defined Types

- `type` can be used to create new names for types
- Example

```
type state = Off | On

let toggle x =
  match x with
  | Off -> On
  | On -> Off
```

Constructing and Destructing Variants

- Syntax: `type t = C1 | ... | Cn`
  - the `Ci` are called constructors and must begin with a capital letter
- Evaluation
  - A constructor `Ci` is already a value
  - Destructing a value `v` of type `t` is done by pattern matching on `v` the patterns are the constructors
- Type Checking
  - `Ci : t` for each `Ci` in `t`'s definition

Data Types: Variants with Data

- Variants can "carry data" too
- Example

```
type shape =
  | Rect of float * float
  | Circle of float
```

- `Rect` and `Circle` are constructors, so a `shape` is either
  - `Rect(w,l)` for any floats `w` and `l`
  - `Circle r` for any float `r`

Data Types: Pattern Matching

- Example

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

- Pattern matching deconstructs values and values can be bound
- Data types are also called algebraic data types or tagged unions

Option Type

```
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"
```

Polymorphic Option Type

- A polymorphic version of `option` type can work with any kind of data

```
type 'a option =
| None
| Some of 'a
```

  - The `'a` is a polymorphic parameter
- This type is built in to OCaml

Recursive Data Types

- We can make our own list type

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

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

len (Cons (1, Cons (2, Cons (3, Nil))))
```

Variants Full Definition

- Syntax: `type t = C1 [of t1] | ... | Cn [of tn]`
  - the `Ci` are called constructors and must begin with a capital letter; may include associated data
- Evaluation
  - A constructor `Ci` is a value if it has no associated data
  - Destructing a value `v` of type `t` is done by pattern matching on `v` the patterns are the constructors with possible data components
- Type Checking
  - `Ci [vi] : t` if `vi` has type `ti`

OCaml Exceptions

```
exception My_exception of in

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" n
```

OCaml Exceptions: Details

- Exceptions are declared with exception
- Exceptions may take arguments
  - Just like type constructors
- Catch exceptions with `try ... with`
  - Pattern-matching can be used in `with`
  - If an exception is not caught
    - the current function exits immediately
    - control transfers up the call chain
    - until the exception is caught or until is reaches the top level

OCaml Common Exceptions

- `failwith s`: raises exception `Failure s`
- `Not_found`: raised by library functions if the object does not exist
- `invalid_arg s`: raises exception `Invalid_argument s`