Algebraic Data Types

Prof. Clarkson
Summer 2015

Today’s music: Union by The Black Eyed Peas (feat. Sting)
Review

Last class:
• Ways to define your own data types: records, tuples, variants, options

Today:
• Algebraic data types
• Error handling with exceptions
• Polymorphism
Type synonyms

Syntax: `type id = t`

- Anywhere you write `t`, you can also write `id`
- The two names are synonymous

E.g.

```plaintext
type point  = float * float
type vector = float list
type matrix = float list list
```
type point = float*float

let getx : point -> float =
    fun (x,_) -> x

let pt : point = (1.,2.)
let floatpair : float*float = (1.,3.)

let one = getx pt
let one' = getx floatpair
ALGEBRAIC DATA TYPES
Recall: Variants

\texttt{type} \ day = \ Sun \ | \ Mon \ | \ Tue \ | \ Wed \\
\hspace{1cm} | \ Thu \ | \ Fri \ | \ Sat

\texttt{type} \ ptype = \ TNormal \ | \ TFire \ | \ TWater

\texttt{type} \ peff = \ ENormal \ | \ ENotVery \ | \ Esuper

So far, just enumerated sets of values
But they can do much more...
Variants that carry data

**type** shape =
  | Point of point
  | Circle of point * float (* center and radius *)
  | Rect of point * point (* lower-left and upper-right corners *)

```ocaml
def pi = acos (-1.0)

let area = function
  | Point _ -> 0.0
  | Circle (_,r) -> pi *. (r ** 2.0)
  | Rect ((x1,y1),(x2,y2)) ->
    let w = x2 -. x1 in
    let h = y2 -. y1 in
    w *. h

let center = function
  | Point p -> p
  | Circle(p,_ ) -> p
  | Rect ((x1,y1),(x2,y2)) ->
    ((x2 -. x1) /. 2.0, (y2 -. y1) /. 2.0)
```
Variants that carry data

```haskell
type shape =
    | Point of point
    | Circle of point * float
    | Rect of point * point
```

Every value of type `shape` is made from exactly one of the constructors and contains:

- a *tag* for which constructor it is from
- the data *carried* by that constructor

Called an *algebraic data type* because it contains product and sum types
Variant types

Type definition syntax:
```
type t = C1 [of t1] | ... | Cn [of tn]
```

A constructor that carries data is *non-constant*

A constructor without data is *constant*
Non-constant variant expressions

Syntax: \( C \ e \)

Evaluation:
if \( e \rightarrow v \) then \( C \ e \rightarrow C \ v \)

Type checking:
\( C \ e : t \)
if \( t = \ldots \mid C \ of \ t' \mid \ldots \)
and \( e : t' \)
Constant variant expressions

Syntax:  C

Evaluation:  already a value

Type checking:
C : t
if t = ... | C | ...
Pattern matching

• Match against constant variants: \( C \)
  (Already had this pattern from last class)

• Match against non-constant variants: \( C \ p \)
  (new today)
RECURSIVE TYPES
Implement lists with variants

```ocaml
type intlist = Nil | Cons of int * intlist

let emp = Nil

let l3 = Cons (3, Nil) (* 3::[ ] or [3] *)

let l123 = Cons(1, Cons(2, l3)) (* [1;2;3] *)

let rec sum (l:intlist) =
    match l with
    | Nil -> 0
    | Cons(h,t) -> h + sum t
```
Implement lists with variants

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

(* length : intlist -> int *)

let empty = function
  | Nil -> true
  | Cons _ -> false

(* empty: intlist -> bool *)
```
Implement lists with variants

```plaintext
let rec fold_right f l acc =
  match l with
  | Nil -> acc
  | Cons(h,t) -> f h (fold_right f t acc)

(* fold_right: (*
  (int -> 'a -> 'a)
  -> intlist -> 'a -> 'a *)

let sumr l = fold_right (+) l 0
(* empty: intlist -> int *)
```
Implement lists with variants

```ocaml
let hd = function
  | Nil      -> ???
  | Cons(h,t) -> h
```

One possibility is to return an option:

```ocaml
let hd = function
  | Nil      -> None
  | Cons(h,t) -> Some h

(* hd: intlist -> int option *)
```

But the standard library throws an exception...
EXCEPTIONS
Example: implement hd

```ocaml
let hd = function
    | Nil -> raise (Failure "empty")
    | Cons(h,t) -> h

# hd Nil;;
Exception: (Failure empty).

let head_or_zero lst =
    try hd lst with
    | Failure s -> 0

# head_or_zero Nil;;
- : int = 0
```
Exceptions: Syntax

Definition:
exception E
exception E of t

Raise (aka throw):
raise e

Catch (aka handle):
try e with
| p1 -> e1
| ...
| pn -> en
Exceptions in standard library

`exception Invalid_argument of string`
raised by library functions to signal that the given arguments do not make sense

`exception Failure of string`
raised by library functions to signal that they are undefined on the given arguments

Convenience function in library:
```
let failwith : string -> 'a =
  fun s -> raise (Failure s)
```
Exceptions: Evaluation

Raise:
If $e \Rightarrow v$ then `raise e` produces an exception packet containing $v$ that propagates upward through the call stack to a handler.

Catch:
```
try e with p1 -> e1 | ... | pn -> en
```
If $e \Rightarrow v$ then the `try` expression evaluates to $v$.
If evaluation of $e$ produces an exception packet, behave like a pattern match on the value in that packet. But if none of the patterns matches, re-raise the exception, thus propagating it upwards.
Exception: Type checking

New kind of type: `exn`
if $E$ is defined as `exception E` then $E : exn$
if $E$ is defined as `exception E of t` and $e : t$
then $E e : exn$

Raise:
if $e : exn$ then `raise e` may have any type $t$

Catch:
if $e$ and $p_1..p_n$ and $e_1..e_n$ all have type $t$
then `try e with $p_1 -> e_1 | ... | p_n -> e_n` has type $t$
POLYMORPHIC VARIANTS
Lists of any type

• **Have:** lists of ints

• **Want:** lists of ints, string, pairs, records, ...

**Non-solution:** copy code

```plaintext
type stringlist = SNil | SCons of string * stringlist
let empty = function
  | SNil -> true
  | SCons _ -> false
```
Lists of any type

Solution: parameterize types on other types

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

let l3 = Cons (3, Nil) (* [3] *)
let lhi = Cons ("hi", Nil) (* "hi" * )
```

**mylist** is not a type but a type constructor: takes a type as input and returns a type

- **int** mylist
- **string** mylist
- `(int*string)` mylist
- ...

Functions on parameterized variants

let rec length = function
  | Nil -> 0
  | Cons (_,t) -> 1 + length t
(* length : 'a mylist -> int *)

let empty = function
  | Nil -> true
  | Cons _ -> false
(* empty: 'a mylist -> bool *)

code stays the same; only the types change
Parametric polymorphism

- *poly* = many, *morph* = form (i.e., shape)
- write function that works for many arguments regardless of their type
- closely related to Java generics, related to C++ template instantiation, ...
THE POWER OF VARIANTS
Lists are just variants

OCaml effectively codes up lists as variants:

```
type 'a list = [] | :: of 'a * 'a list
```

- Just a bit of syntactic magic in the compiler to use `[]` and `::` instead of alphabetic identifiers
- `[]` and `::` are constructors
- `list` is a type constructor parameterized on type variable `'a`
Options are just variants

OCaml effectively codes up options as variants:

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

- **None** and **Some** are constructors
- **option** is a type constructor parameterized on type variable `'a`
Exceptions are weird variants

• Think of **exn** as a variant type
• An exception definition `exception E [of t]` adds a new constructor to that variant
  – can't do that with normal variants
• **Build** an exception value by writing an expression with that constructor
  – like normal variants
• **Use** an exception value to transfer control using `raise` and `try`
  – can't do that with normal variants
• **Destruct** an exception value by pattern matching
  – like normal variants
TREES
Binary trees

type 'a bintree =
    | Leaf
    | Node of 'a bintree * 'a * 'a bintree

let t =
    Node(
        Node( Leaf, 1, Leaf ),
        2,
        Node( Leaf, 3, Leaf )
    )
Binary trees

let rec size = function
   | Leaf -> 0
   | Node(l,_,r) -> (size l) + 1 + (size r)

let rec depth = function
   | Leaf -> 0
   | Node(l,_,r) -> 1 + max (depth l) (depth r)

let rec sum = function
   | Leaf -> 0
   | Node(l,x,r) -> (sum l) + x + (sum r)