CS 3110

Variants

Prof. Clarkson
Fall 2015

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

Review

Previously in 3110:

- User-defined data types: records, tuples, variants
- Built-in data types: lists, options

Today:

- More about variants
- Polymorphism
- Exceptions

Variants vs. records vs. tuples

	Define	Build/construct	Access/destruct
Variant	`type`	Constructor name	Pattern matching
Record	`type`	Record expression with `{…}`	Pattern matching OR field selection with dot operator `.`
Tuple	N/A	Tuple expression with `(…)`	Pattern matching OR `fst` or `snd`

- Variants: **one-of types** *aka* **sum types**
- Records, tuples: **each-of types** *aka* **product types**

Question

Which of the following would be better represented with records rather than variants?

A. *Coins*, which can be pennies, nickels, dimes, or quarters

B. *Students*, who have names and id numbers

C. A *dessert*, which has a sauce, a creamy component, and a crunchy component

D. A and C

E. B and C

Question

Which of the following would be better represented with records rather than variants?

A. Coins, which can be pennies, nickels, dimes, or quarters

B. Students, who have names and NetIDs

C. A *dessert*, which has a sauce, a creamy component, and a crunchy component

D. A and C

E. B and C

TYPE SYNONYMS

Type synonyms

Syntax: **`type id = t`**

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

e.g.

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

Type synonyms

```
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
```

VARIANTS

Recall: Variants

```
type day = Sun | Mon | Tue | Wed
         | Thu | Fri | Sat

type ptype = TNormal | TFire | TWater

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 *)

let 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

```
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 ==> v` then `C e ==> C v`

Type checking:

`C e : t`

if `t = ... | C of t' | ...`

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

```
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

```
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 *)
```

PARAMETERIZED VARIANTS

Lists of any type

- **Have:** lists of ints
- **Want:** lists of ints, string, pairs, records, ...

Non-solution: copy code

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

Lists of any type

Solution: parameterize types on other types

```
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:

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

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

EXCEPTIONS

Example: implement hd

```
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 ==> 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 ==> 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 `e1..en` all have type `t`
and `p1..pn` all have type `exn`
then `try e with p1 -> e1 | ... | pn -> en`
has type `t`

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
  - possible to do that with normal variants, but not recommended
- **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

A2

- Out now on course website, due in about 9 days
- Implement a text adventure game engine, and write your own adventure
- Need trees and polymorphic variants: will see in recitation tomorrow
- Suggestion: start early, give plenty of thought to design and testing

Upcoming events

- [today] A2 out

This is variant.

THIS IS 3110