More Variants

Nate Foster
Spring 2018
Review

Previously in 3110:
• User-defined data types: records, variants
• Built-in type constructors: list, option

Today:
• * (i.e., Tuple Types)
• Type synonyms
• More about variants
• Exceptions
Tuples

- Several pieces of data glued together
- A tuple contains several components
- (Don't have to define tuple type before use)

E.g.,
- \((1,2,10)\)
- \(1,2,10\)
- \((\text{true}, \ "Hello")\)
- \(([1;2;3], (0.5,'X'))\)
Tuple types

- (1,2,10) : int*int*int
- 1,2,10 : int*int*int
- (true, "Hello") : bool*string
- ([1;2;3], (0.5,'x')) : int list * (float*char)
Tuples

- 2-tuple: pair
- 3-tuple: triple
- beyond that: maybe better to use records

We need language constructs to *build* tuples and to *access* the components
- Building is easy: just write the tuple, as before
- Accessing uses pattern matching...
Accessing tuples

New kind of pattern, the **tuple pattern**: \((p_1, \ldots, p_n)\)

```plaintext
match \((1,2,3)\) with
  | \((x,y,z)\) => x+y+z

(* => 6 *)
```

```plaintext
let thrd t =
  match t with
  | \((x,y,z)\) => z

(* thrd : 'a*'b*'c => 'c *)
```

Note: we never needed more than one branch in the match expression...
Pattern matching without match

(* OK *)
let thrd t =
  match t with
  | (x,y,z) -> z

(* good *)
let thrd t =
  let (x,y,z) = t in z

(* better *)
let thrd t =
  let (_,_,z) = t in z

(* best *)
let thrd (_,_,z) = z
Variants vs. records vs. tuples

<table>
<thead>
<tr>
<th></th>
<th>Define</th>
<th>Build</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variant</td>
<td><strong>type</strong></td>
<td>Constructor name</td>
<td>Pattern matching</td>
</tr>
<tr>
<td>Record</td>
<td><strong>type</strong></td>
<td>Record expression with {...}</td>
<td>Pattern matching OR field selection with dot operator .</td>
</tr>
<tr>
<td>Tuple</td>
<td>N/A</td>
<td>Tuple expression with (...)</td>
<td>Pattern matching OR \texttt{fst} or \texttt{snd}</td>
</tr>
</tbody>
</table>

• 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

\texttt{type \ day = Sun | Mon | Tue | Wed}
\texttt{  | 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 *)

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
Variants that carry data

```haskell
{var} shape =
  | Point of point
  | Circle of point * float
  | Rect of point * point

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

def 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, aka *tagged union*
Tagged union

- *Union* because the set of all values of the type is the union of the set of all values of the individual constructors
  
  \[
  \text{type } t = \text{String of string} \\
  \quad \mid \text{Int of int}
  \]

- *Tagged* because possible to determine which underlying set a value came from

  \[
  \text{type } t = \text{Left of int} \\
  \quad \mid \text{Right of int}
  \]

- "All for one and one for all":
  - all values of variant, regardless of constructor, have same type
  - any one value of variant built with exactly one constructor, all of which are specified in type definition
Variant types

Type definition syntax:
\[
\text{type } t = C_1 \text{ [of } t_1] \mid \ldots \mid C_n \text{ [of } t_n]\]

A constructor that carries data is \textit{non-constant}
A constructor without data is \textit{constant}

Semantics are straightforward; see notes
Given our shape variant, which function would determine whether a shape is a circle centered at the origin?

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

Possible answers on next slide...
**type** shape = Point of point | Circle of point * float | Rect of point * point

let cato = function |
Point -> false |
Circle -> true |
Rect -> false

let cato = function |
Point _ | Rect _ -> false |
Circle (0.,0.), r -> true

let cato c =
c = Circle ((0.,0.),_)

let cato = function |
Point p -> false |
Circle (0.,0.) -> true |
Rect (ll,ur) -> false

let cato = function |
Circle ((0.,0.), _) -> true |
_ -> false
type shape = Point of point | Circle of point * float | Rect of point * point

let cato = function
| Point -> false
| Circle -> true
| Rect -> false

let cato = function
| Point _ | Rect _ -> false
| Circle (0.,0.), r -> true

let cato c =
c = Circle ((0.,0.),_)

let cato = function
| Point p -> false
| Circle (0.,0.) -> true
| Rect (ll,ur) -> false

let cato = function
| Circle ((0.,0.), _) -> true
| _ -> false
RECURSIVE VARIANTS
Implement lists with variants

```ocaml
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 *)
PARAMETERIZED VARIANTS
Lists of any type

- **Have:** lists of ints
- **Want:** lists of ints, lists of strings, lists of pairs, lists of records that themselves contain lists of pairs, ...

**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

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

let l3 = Cons (3, Nil) (* [3] *)
let lhi = Cons ("hi", Nil) (* ["hi"] *)
Lists of any type

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

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
• 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
```

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

OCaml effectively codes up options as variants:

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

- **option** is a type constructor parameterized on type variable 'a
- **None** and **Some** are constructors
Exceptions are (mostly) just variants

OCaml effectively codes up exceptions as slightly strange variants:

```ocaml
type exn
exception MyNewException of string
```

- Type `exn` is an *extensible* variant that may have new constructors added after its original definition
- Raise exceptions with `raise e`, where `e` is a value of type `exn`
- Handle exceptions with pattern matching, just like you would process any variant
Upcoming events

• [by Fri morning] A1 out
• [Feb 12 @ 5:30] WICC/ACSU/URMC G-Body on CIS Culture
• [Feb 12 @ 6:30] WICC Partner Finding Social