

CSE 130 : Fall 2011

Programming Languages

Lecture 3: Data Types

Ranjit Jhala
UC San Diego



Recap from last time

A shorthand for function binding

```
# let neg = fun f x -> fun x -> not (f x);  
...  
# let neg f x = not (f x);  
val neg : ('a -> bool) -> 'a -> bool = fn  
  
# let is5gte = neg is5lt;  
val is5gte : int -> bool = fn;  
  
# is5gte 10;  
val it : bool = false;  
  
# is5gte 2;  
val it : bool = true;
```

Put it together: a “filter” function

If arg “matches”
this pattern... ...then use
this Body Expr

```
- let rec filter f l =  
  match l with  
  | [] -> []  
  | (h::t) -> if f h  
               then h::(filter f t)  
               else (filter f t);;  
  
val filter : ('a->bool)->'a list->'a list) = fn  
  
# let list1 = [1;31;12;4;7;2;10];;  
# filter is5lt list1 ;;  
val it : int list = [31;12;7;10]  
# filter is5gte list1;;  
val it : int list = [1;4;2]  
# filter even list1;;  
val it : int list = [12;4;2;10]
```

Put it together: a “partition” function

```
# let partition f l = (filter f l, filter (neg f) l);
val partition : ('a->bool)->'a list->'a list*'a list = fn

# let list1 = [1;31;12;4;7;2;10];
- ...
# partition is5lt list1 ;
val it : (int list * int list) = ([31;12;7;10],[1;2;10])

# partition even list1;
val it : (int list * int list) = ([12;4;2;10],[1;31;7])
```

A little trick ...

```
# 2 <= 3;; ...
val it : bool = true
# "ba" <= "ab";;
val it : bool = false

# let lt = (<) ;;
val it : 'a -> 'a -> bool = fn

# lt 2 3;;
val it : bool = true;
# lt "ba" "ab" ;;
val it : bool = false;

# let is5lt = lt 5;
val is5lt : int -> bool = fn;
# is5lt 10;
val it : bool = true;
# is5lt 2;
val it : bool = false;
```

Tuesday, October 4, 2011

5

Tuesday, October 4, 2011

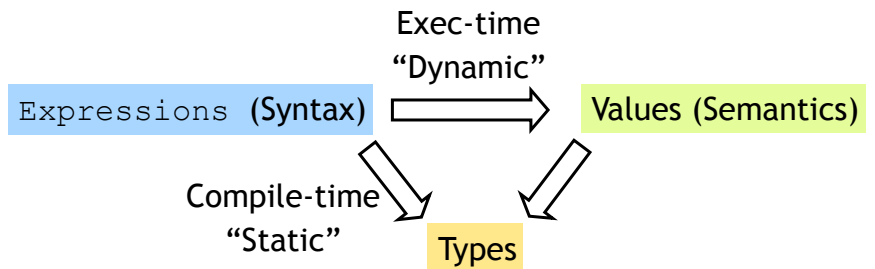
6

Put it together: a “quicksort” function

```
let rec sort xs =
  match xs with
  | [] -> []
  | (h::t) ->
    let (l,r) = partition ((<) h) t in
    (sort l)@(h::(sort r))
;;
```

Now, lets begin at the beginning ...

Begin at the beginning ...



1. Programmer enters expression
2. ML checks if expression is “well-typed”
 - Using a precise set of rules, ML tries to find a unique type for the expression meaningful type for the expr
3. ML evaluates expression to compute value
 - Of the same “type” found in step 2

Tuesday, October 4, 2011

7

Tuesday, October 4, 2011

8

Today: Different Types of Data

- We've seen some **base** types and values:
 - Integers, Floats, Bool, String etc.
- Some ways to **build** up types:
 - Products (tuples), records, "lists"
 - Functions

9

In Class Exercise #1

10

Tuesday, October 4, 2011

9 Tuesday, October 4, 2011

10

What About More Complex Data ?

- We've seen some **base** types and values:
 - Integers, Floats, Bool, String etc.
- Some ways to **build** up types:
 - Products (tuples), records, "lists"
 - Functions
- Design Principle: **Orthogonality**
 - Don't clutter **core language** with stuff
 - Few, powerful orthogonal building techniques
 - Put "**derived**" types, values, functions in **libraries**

Tuesday, October 4, 2011

11 Tuesday, October 4, 2011

How To Build Datatypes

Three key ways to build complex types/values

1. "Each-of" types

Value of T contains value of T1 **and** a value of T2

2. "One-of" types

Value of T contains value of T1 **or** a value of T2

3. "Recursive"

Value of T contains (sub)-value of **same type** T

12

Suppose I wanted ...

... a program that processed lists of attributes

- Name (string)
- Age (integer)
- DOB (int-int-int)
- Address (string)
- Height (float)
- Alive (boolean)
- Phone (int-int)
- email (string)

Many kinds of attributes:

- too many to put in a record
- can have multiple names, addresses, phones, emails etc.

Constructing Datatypes

```
type t = c1 of t1 | c2 of t2 | ... | Cn of tn
```

`t` is a new datatype.

A value of type `t` is either:

a value of type `t1` placed in a box labeled `c1`

Or a value of type `t2` placed in a box labeled `c2`

Or ...

Or a value of type `tn` placed in a box labeled `Cn`

Suppose I wanted ...

Attributes:

- Name (string)
- Age (integer)
- DOB (int-int-int)
- Address (string)
- Height (real)
- Alive (boolean)
- Phone (int-int)
- email (string)

```
type attrib =  
  Name of string  
| Age of int  
| DOB of int*int*int  
| Address of string  
| Height of float  
| Alive of bool  
| Phone of int*int  
| Email of string;;
```

Creating Values

How to create values of type `attrib`?

```
# let a1 = Name "Ranjit";;  
val x : attrib = Name "Ranjit"  
# let a2 = Height 5.83;;  
val a2 : attrib = Height 5.83  
# let year = 1977 ;;  
val year : int = 1977  
# let a3 = DOB (9,8,year) ;;  
val a3 : attrib = DOB (9,8,1977)  
# let a_list = [a1;a2;a3];;  
val a_list : attrib list = ...
```

```
type attrib =  
  Name of string  
| Age of int  
| DOB of int*int*int  
| Address of string  
| Height of float  
| Alive of bool  
| Phone of int*int  
| Email of string;;
```

One-of types

- We've defined a "one-of" type named `attrib`

- Elements are one of:

- string,
- int,
- int*int*int,
- float,
- bool ...

```
datatype attrib =  
  Name of string  
| Age of int  
| DOB of int*int*int  
| Address of string  
| Height of real  
| Alive of bool  
| Phone of int*int  
| Email of string;
```

- Can create uniform `attrib` lists

- Suppose I want a function to print attribs...

How to tell whats in the box ?

```
type attrib =  
  Name of string  
| Age of int  
| DOB of int*int*int  
| Address of string  
| Height of float  
| Alive of bool  
| Phone of int*int  
| Email of string;;
```

```
match e with  
  Name s -> e1  
| Age i -> e2  
| DOB (m,d,y) -> e3  
| Address addr -> e4  
| Height h -> e5  
| Alive b -> e6  
| Phone (a,n) -> e7  
| Email e -> e8
```

Pattern-match expression: check if e is of the form ...

- On match:

- value in box bound to pattern variable
- matching result expression is evaluated
- Simultaneously test and extract contents of box

match-with is an Expression

```
match e with  
  Name s -> e1  
| Age i -> e2  
| DOB (m,d,y) -> e3  
| Address addr -> e4  
| Height h -> e5  
| Alive b -> e6  
| Phone (a,n) -> e7  
| Email e -> e8
```

match-with is an Expression

```
match e with  
  C1 x1 -> e1  
| C2 x2 -> e2  
| ...  
| Cn xn -> en
```

Type rules ?

- *e1, e2, ..., en* must have same type
- Which is type of whole expression

Benefits of match-with

```
match e with
  C1 x1 -> e1
| C2 x2 -> e2
| ...
| Cn xn -> en

type t =
  C1 of t1
| C2 of t2
| ...
| Cn of tn
```

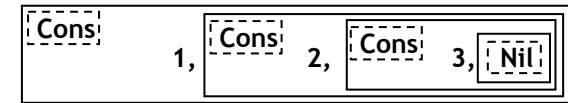
1. Simultaneous test-extract-bind
2. Compile-time checks for:
 - missed cases: ML warns if you miss a t value
 - redundant cases: ML warns if a case never matches

What about “Recursive” types ?

```
type int_list =
  Nil
| Cons of int * int_list
```

Think about this! What are values of `int_list` ?

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



Lists aren't built-in!

```
type int_list =
  Nil
| Cons of int * int_list
```

Lists are **derived** using an elegant core!

1. Each-of
2. One-of
3. Recursive

`::` is just a pretty way to say “Cons”
`[]` is just a pretty way to say “Nil”

Some functions on Lists: Length

```
let rec len l =
  match l with
  Nil -> 0
| Cons(h,t) -> 1 + (len t)
```

Base pattern (Nil) and *Ind pattern* (Cons(h,t)) are highlighted. *Base Expression* (0) and *Inductive Expression* (1 + (len t)) are also highlighted.

```
let rec len l =
  match l with
  Nil -> 0
| Cons(_,t) -> 1 + (len t)
```

Matches everything, no binding

```
let rec len l =
  match l with
  Cons(_,t) -> 1 + (len t)
| _ -> 0
```

Pattern-matching in order
- Must match with Nil

Some functions on Lists : Append

```
let rec append (l1,l2) =
```

Base pattern

Base Expression

Ind pattern

Inductive Expression

- Find the right **induction** strategy
 - Base case: pattern + expression
 - Induction case: pattern + expression

Well designed datatype gives strategy

null, hd, tl are all functions ...

Bad ML style: More than aesthetics !

Pattern-matching better than test-extract:

- ML checks **all cases covered**
- ML checks **no redundant cases**
- ...at **compile-time**:
 - fewer errors (crashes) during execution
 - get the bugs out ASAP!

Another Example: Calculator

We want an arithmetic calculator to evaluate expressions like:

- $4.0 + 2.9 = 6.9$
- $3.78 - 5.92 = -2.14$
- $(4.0 + 2.9) * (3.78 - 5.92) = -14.766$

Q: Whats a ML datatype for such expressions ?

Another Example: Calculator

We want an arithmetic calculator to evaluate expressions like:

- $4.0 + 2.9 = 6.9$
- $3.78 - 5.92 = -2.14$
- $(4.0 + 2.9) * (3.78 - 5.92) = -14.766$

Whats a ML function for **evaluating** such expressions ?

