

CSE 130: Fall 2011

Programming Languages

Lecture 8: Polymorphism (contd)

Ranjit Jhala
UC San Diego



News

- PA3 Deadline Moved
 - Friday → Monday 5pm
- PA4 will be up by tomorrow
 - Essential for midterm...

Polymorphism enables Reuse

- Can reuse generic functions:

```
map : 'a * 'b -> 'b * 'a
```

```
filter: ('a -> bool) -> 'a list -> 'a list
```

```
rev: 'a list -> 'a list
```

```
length: 'a list -> int
```

```
swap: 'a * 'b -> 'b * 'a
```

```
sort: ('a -> 'a -> bool) -> 'a list -> 'a list
```

```
fold: ...
```

- If function (algorithm) is independent of type, can reuse code for all types !

Polymorphic Data Types

- Data types are also polymorphic!

```
type 'a list =  
  Nil  
| Cons of ('a * 'a list)
```

- Type is instantiated for each use:

Cons(1,Cons(2,Nil)) :

Cons("a",Cons("b",Nil)) :

Cons((1,2),Cons((3,4),Nil)) :

Nil :

Polymorphic Data Types

- Data types are also polymorphic!

```
type 'a list =  
  Nil  
| Cons of ('a * 'a list)
```

- Type is instantiated for each use:

Cons(1,Cons(2,Nil)) : int list

Cons("a",Cons("b",Nil)) : string list

*Cons((1,2),Cons((3,4),Nil)) : (int*int) list*

Nil : 'a list

Options

- Consider the type:

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

- Why is this useful?

Remember This Guy?

```
val assoc: 'b -> 'a -> ('a * 'b) list -> 'b
```

We had to pass in “default” value (yuck!)

Instead, use “option” to return failure...

```
val assoc: 'a -> ('a * 'b) list -> 'b option
```

Example: Calculator Revisited

```
type expr =  
  | Num of int  
  | Div of expr * expr
```

Can you write a function?

```
val eval : expr -> int
```

In Class Exercise

```
type expr =  
  | Num of int  
  | Div of expr * expr
```

Write an Evaluation function

```
val eval : expr -> int option
```

That returns *None* if a div-by-zero occurs

Datatypes with many type variables

- Multiple type variables

```
type ('a,'b) tree =  
  Leaf of ('a * 'b)  
| Node of ('a,'b) tree * ('a,'b) tree
```

- Type is instantiated for each use:

Leaf("joe",1) :

Leaf("william",2) :

Node(...,...) :

Node(Leaf("joe",1),Leaf(3.14, "pi")):

Datatypes with many type variables

- Multiple type variables

```
type ('a,'b) tree =  
  Leaf of ('a * 'b)  
| Node of ('a,'b) tree * ('a,'b) tree
```

- Type is instantiated for each use:

Leaf("joe",1) : (string,int) tree

Leaf("william",2) : (string,int) tree

Node(...,...) : (string,int) tree

Node(Leaf("joe",1),Leaf(3.14, "pi"))

In-Class Exercise!

- Multiple type variables

```
type ('a, 'b) tree =  
  Leaf of ('a * 'b)  
| Node of ('a, 'b) tree * ('a, 'b) tree
```

Write a function to lookup keys...

```
val lookup: 'a -> ('a, 'b) tree -> 'b option
```

Binary Search Trees

```
type ('a, 'b) bst =  
  Leaf  
| Node of ('a * 'b * ('a, 'b) bst * ('a, 'b) bst)
```

Node (key, value, left, right)

Property:

keys in left < key < keys in right

Binary Search Trees

```
type ('a, 'b) bst =  
  Leaf  
| Node of 'a * 'b * ('a, 'b) bst * ('a, 'b) bst
```

Now we can write an efficient lookup

```
val lookup: 'a -> ('a, 'b) bst -> 'b option
```

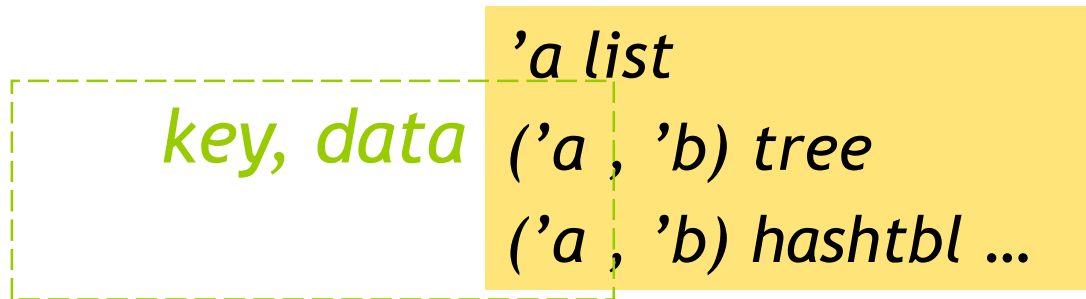
A trick question: consider this type

```
type ('a, 'b) list =  
  | Nil  
  | Cons of 'a * ('b * 'a) list
```

Can you describe some values of this type?

Polymorphic Data Structures

- Container data structures independent of type !
- Appropriate type is **instantiated** at each **use**:



- Appropriate type instantiated at use
 - No **unsafe casting** as in C++/Java
- **Static type checking catches errors** early
 - Cannot add **int** key to **string** hashtable
- **Generics**: in Java, C#, VB (borrowed from ML)