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!

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

- Type is instantiated for each use:

```ml
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:

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

- Why is this useful?

Remember This Guy?

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

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

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

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

Example: Calculator Revisited

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

Can you write a function?

```ml
val eval : expr -> int
```
In Class Exercise

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

Write an Evaluation function

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

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

---

Datatypes with many type variables

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

Let 

```ml
let x = Node ("alice", 5, Leaf, Leaf)
```

Q: What is the type of \(x\) ?

(a) \((\text{int}, \text{string})\) tree
(b) \((\text{\('a', 'b')}\) tree
(c) int tree
(d) string tree
(e) \((\text{string, int})\) tree
Datatypes with many type variables

- Multiple type variables

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

- Type is instantiated for each use:

  ```
  Node("alice", 2, Leaf, Leaf)
  Node("charlie", 3, Leaf, Leaf)
  Node("bob", 13,
       , Node("alice", 2, Leaf, Leaf)
       , Node(3, "charlie", Leaf, Leaf))
  ```

Binary Search Trees

BST Property:

- keys in left < key < keys in right

  ```
  Node (key, value, left, right)
  ```

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

In-Class Exercise!

Write a function to lookup keys...

```
val lookup: 'a ->('a,'b)tree ->'b option
```
A tricky question: consider this type

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

Which is a valid Ocaml Expression?

(a) Cons(1, Cons("a", Cons(3.14, Nil)))
(b) Cons(1, Cons("a", Cons(1, Nil)))
(c) Cons(1, Cons("a", Cons("a", Nil)))
(d) Cons(1, Cons(1, Cons("a", Nil)))
(e) Cons(1, Cons(1, Cons(1, Nil)))

Polymorphic Data Structures

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

  ```
  'a list
  ('a , 'b) tree
  ('a , 'b) hashtbl ...
  ```

- Static type checking catches errors early
  - Cannot add int key to string hashtable

- Generics: in Java, C#, VB (borrowed from ML)

Polymorphic Types

- Polymorphic types are tricky
- Not always obvious from staring at code
- How to ensure correctness?
- Types (almost) never entered w/ program!

Type Inference

How DOES Ocaml figure out all the types?!
Polymorphic Type Inference

• Computing the types of all expressions
  - At compile time: statically Typed

• Each binding is processed in order
  - Types are computed for each binding
  - For expression and variable bound to
  - Types used for subsequent bindings

• Unlike values (determined at run-time)

Example 1

```ml
let x = 2 + 3;;
let y = string_of_int x;;
```

Example 2

```ml
let x = 2 + 3;;
let y = string_of_int x;;
let inc y = x + y;;
```
What's the type of foo?

```ml
let foo x =
  let (y, z) = x in
  z - y
```

(a) int
(b) int * int
(c) int * int -> int
(d) int -> int -> int
(e) Error

Example 4

```ml
let rec cat xs =
  match xs with
  | []    -> cat []
  | x::xs -> x ^ (cat xs)
```

(a) string -> string
(b) string
(c) string list -> string list
(d) string list -> string
(e) Error

Example 5

```ml
let rec cat xs =
  match xs with
  | []    -> ""
  | x::xs -> x ^ (cat xs)
```

ML doesn’t know what function does, or even that it finishes only its type!
Example 5

```ocaml
let rec map f xs =
  match xs with
  | []     -> []
  | x::xs' ->(f x)::(map f xs')
```

“Generalize” Unconstrained Vars

`('a->'b) -> 'a list -> 'b list`

What is the type of `<+>`

```ocaml
let (<+>) f g x = g (f x)
```

(a) 'a -> 'b -> 'c -> 'd
(b) ('a->'b)->('a ->'b)->('a ->'b)
(c) (int->char)->(char->bool)->(int->bool)
(d) (int->int)->(int->int)->(int->int)
(e) ('a->'b)->('b ->'c)->('a ->'c)

Example 7

```ocaml
let rec fold f cur xs =
  match xs with
    | []     -> cur
    | x::xs' -> fold f (f cur x) xs'
```
Example 11

let fool f g x =
  if f x
  then x
  else g x

Example 12

let foo2 f g x =
  if f x
  then x
  else foo2 f g (g x)