ML Lectures (continued)
Winter 2007

Polymorphism

There are 3 kinds of polymorphism:

1. Ad-hoc polymorphism: also known as *overloading*. Different operations known by same name that the compiler/interpreter resolves.

2. Inheritance-based polymorphism: subclasses define new version of methods possessed by super class. OO languages use this a lot!!

3. Parametric Polymorphism: types/type variables explicitly used as parameters.

Polymorphism

3. Parametric Polymorphism:

- Allows types to be parameters to functions and other types.
- Basic idea is to have a type variable…
- Type of function depend on type of parameter
- Implementation:

  Homogenous implementations (ML)
  - One one copy of code is generated
  - Polymorphic parameters must internally be implemented as pointers

  Heterogeneous implementation (C++)
  - One copy of function code per instantiation
  - Access to polymorphic parameters can be more efficient

Polymorphic Functions

Function Polymorphism:
values (including variables or functions) that can have more than one type

Examples:

```
fun length L = if (null L) then 0 else 1 + length (tl L);

fun reverse [] = []
  | reverse (h::t) = reverse(t) @ [h];

fun listify x = [x];

fun apply (f,x) = (f x);
apply(real,5);
```

Without polymorphism, we would need many functions:
int-length, int-reverse, real-length, real-reverse, etc.

Polymorphism

Greek: *poly = many , morph = form*

Definitions:

Polymorphism:
- dictionary.com: the capability of assuming different forms; the capability of widely varying in form. The occurrence of different forms, stages, or types
- Software: a value/variable can belong to multiple types

Monomorphism:
Dictionary.com: having only one form, same genotype…
Software: every value/variable belongs to exactly one type

Without polymorphism, a typed language would be very rigid.

We would have to define many different kinds of *length* functions:
int-length : int list → int
real-length: real list → int
string-length: string list → int ……..

And the code for each of these functions would be virtually identical!

Polymorphism adds flexibility & convenience.

Polymorphism

1. Ad-hoc polymorphism:

Different operations on different types known by the same name *(also called overloading)*

E.g. 3.0 + 4
compiler/interpreter must change 4 to 4.0 first

2. Inheritance polymorphism:

- Use sub-classing to define new versions of existing functions *(OO)*

E.g.:

```
public class Employee{
        public int salary;
        public void income() = {return
    salary;}
    }
    public class Waitress extends Employee{
        public int tips;
        public void income() = {return
    (salary + tips);}
    public class Professor extends Employee;
```

Parametric Polymorphism Examples

type (<list type params>) <identifier> = <type expr>

Example 1- pair
-type ‘a pair = ‘a * ‘a;
*type ‘a pair = ‘a * ‘a*

-(1,2): int pair;
val it = (1,2): int pair

Example 2- word count
- type (‘d,’r) mapping = (‘d * ‘r) list;
*type (‘a, ‘b) mapping = (‘a * ‘b) list*

-val wc = (“in”,5), (“a”,1)]: (string, int) mapping;
val wc – [(“in”,5), (“a”,1)] : (string, int) mapping

Polymorphic Functions

Polymorphic functions are common in ML:

- fun id X = X;
val id = fn : 'a -> 'a

```
- id 7;
val it = 7 : int
- id "abc";
val it = "abc" : string
```

- fun listify X = [X];
val listify = fn : 'a -> 'a list

```
- listify 3;
val it = [3] : int list
- listify 7.3;
val it = [7.3] : real list
```

- fun double X = (X,X);
*val double = fn : 'a -> 'a * 'a*

```
- double “xy”;
val it = ("xy","xy") : string * string
- double [1,2,3];
val it = ([1,2,3],[1,2,3]) : int list * int list
```

Polymorphic Functions

- fun inc(N,X) = (N+1,X);
*val inc = fn : int * 'a -> int * 'a*

```
- inc (2,5);
val it = (3,5) : int * int
- inc (4,(34,5));
val it = (5,(34,5)) : int * (int * int)
```

- fun swap(X,Y) = (Y,X);
*val swap = fn : 'a * 'b -> 'b * 'a*

```
- swap (“abc”,7);
val it = (7,"abc") : int * string
- swap (13.4,[12,3,3]);
val it = ([12,3,3],13.4) : int list * real
```

- fun pair2list(X,Y) = [X,Y];
*val pair2list = fn : 'a * 'a -> 'a list*

```
- pair2list(1,2);
  val it = [1,2] : int list
- pair2list(1,"cd");
?
```


Polymorphism

Operators that restrict polymorphism

- Arithmetic operators: **+ , -, * , –**
- Division-related operations e.g. **/ , div, mod**
- Inequality comparison operators: **< , <=, >=, >**,etc.
- Boolean connectives: **andalso, orelse, not**
- String concatenation operator: **^**
- Type conversion operators
  - E.g. **ord, chr, real, str, floor, ceiling, round, truncate,…**

Operators that allow polymorphism

- Tuple operators
- List operators
- Equality operators **=, <>**

Polymorphic Functions

- fun apply(Func,X) = Func X;
*val apply = fn : ('a -> 'b) * 'a -> 'b*

```
- apply (hd, [1,2,3]);
val it = 1 : int
- apply (length, [23,100]);
val it = 2 : integer
```

- fun applytwice(Func,X) = Func(Func X);
*val applytwice = fn : ('a -> 'a) * 'a -> 'a*

```
- applytwice (square,3);
val it = 81 : int
- applytwice (tl, [1,2,3,4]);
?
- applytwice (hd, [1,2,3,4]);
?
```

Equality Types and `a versus ``a

= and <> are **equality operators**

ML defines a class of types called equality types, which are types that allow equality to be tested. Most basic types are equality types – integer, boolean, character and string, **not** functions

One can form more equality types by forming tuples or lists of equality types.

If a function uses equality comparison, it restricts the type to an equality type, as illustrated in the examples below.

The following examples are from [Ullman, 1998, pg. 153]

```
1 fun rev1(L) =
2             if L = nil then nl
3             else rev1(tl(L)) @ [hd(L)];
   val rev1 = fun : ''a list -> ''a list
```

Reversal using an equality comparison

```
4 fun rev2(nil) = nil
5 |     rev2(x::xs) = rev2(xs) @ [x]
   val rev2 = fun : 'a list -> 'a list
```

Reversal *without* an equality comparison