Principles of programming languages

Lecture 5

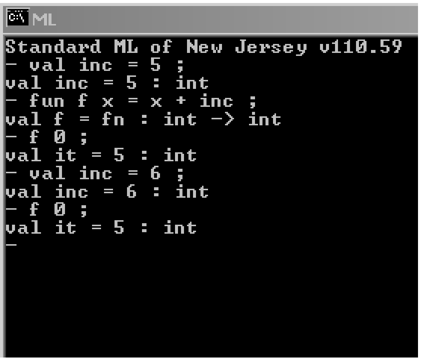
http://few.vu.nl/~nsilvis/PPL/2007

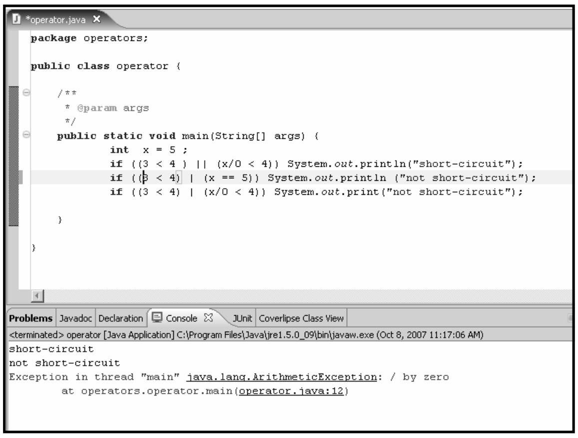
Natalia Silvis-Cividjian
e-mail: nsilvis@few.vu.nl

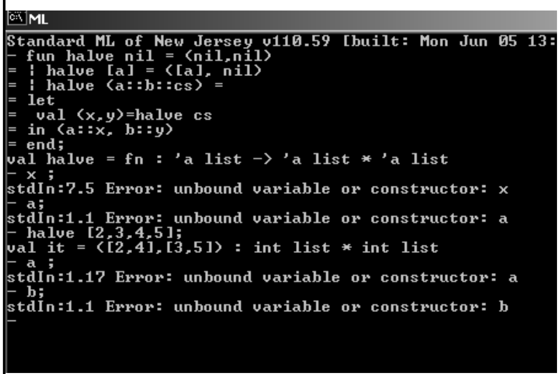
vrije Universiteit *amsterdam*

From last week

- ML has no assignment. Explain how to access an old binding ?
- Is & for logical and? If both operands are boolean, then & can be used as && but without short-circuiting
- ML: Value of a in halve?





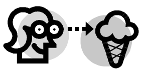


Outline

Part I. Polymorphism

Part II. Functional programming. Third look at ML.

What is polymorphism ?



Polymorphism

- `poly+morphos` (Greek) = many forms (shapes)
- Used in many other sciences: material science, biology
- In programming languages PMF is difficult to define:
  - Applies to a wide variety of language features
  - Most languages have at least a little
  - First we examine a few major examples, then we try to give a definition that covers them

Polymorfism examples

- **Overloading**
- **Parameter coercion**
- **Parametric polymorphism**
- **Definitions and classifications**

What is overloading?

Overloading = to give more than one definition, all of different types

You can overload:

operators

or

function names

Predefined Overloaded **Operators**

Some operators are already overloaded by the language itself.

ML:
```
val x = 1 + 2;
val y = 1.0 + 2.0;
```

C++:
```
a = 1 + 2;
b = 1.0 + 2.0;
c = "hello " + "there";
```

How does the languages system know which definition to use?

User-Overloaded Operators

Situations:

You can add two integers or two floats, but what if you want to add two complex numbers ?

You can compare two integers, two strings, but how to compare 2 C structures?

Ex: Overloading the < **operator in C++**

```
#include <iostream>
#include <string>

using namespace std ;

struct Student
{
       string name ;
       int stud_nr ;
       int grade ;
};

// overloads < operator to compare 2 structs of type Student

bool operator< (const Student& student1, const Student& student2)
{
       if (student1.grade < student2.grade) return true ;
       else return false ;
}
```

Ex: Overloading the < operator in C++ (ctnd)

```
int main()
{
  Student me, you ;

  me.name = "Sheila" ;
  me.stud_nr =14537780 ;
  me.grade = 8 ;

  you.name = "Bob" ;
  you.stud_nr =14532180 ;
  you.grade = 4 ;

  if (me < you) cout << you.name << " is more clever than " <<
me.name ;
  else cout << me.name << " is more clever than " << you.name ;
  cout << endl ;

  return 0 ;
}
```

Output: Sheila is more cleaver than Bob

C++ Operator overloading rules

•All operators can be overloaded except:
. (direct member), :: (scope resolution), .* and ?:

•You cannot change the unary/binary nature of an operator.

•You cannot override precedence rules.

Take care! Overloading can become confusing. The user has to use his common sense and not overdo it.

Overloaded Function Names

- In some languages (Java, C++), the user can overload the function name = same name, **different** definition (different semantics)

```
int square(int x) {
  return x*x;
}

double square(double x) {
  return x*x;
}
```

The bodies look the same, but the implementation by hardware is very different.

Implementing Overloading

- Compilers implement overloading (solve the ambiguity) like this:
  - Create a set of monomorphic functions, one for each definition
  - Invent a **mangled** name for each, encoding the type information Ex: fred_Fii
  - Have each reference use the appropriate mangled name, depending on the **parameter types**

Main feature of overloading

The language system looks at the operators type and decides which definition to use.

Polymorphism examples

- **Overloading**
- **Parameter coercion**
- **Parametric polymorphism**
- **Definitions and classifications**

Coercion

A coercion is an **implicit** type conversion, supplied automatically

Explicit type conversion in Java:

```
double x;
x = (double) 2;
```

OR

Implicit conversion (Coercion) in Java:

```
double x;
x = 2;
```

This coercion is not polymorphism, x is not polymorphic

Defining Coercions

- Language definitions often take many pages to define exactly which coercions are performed
- Some languages, especially some older languages like Algol 68 and PL/I, have very extensive powers of coercion
- Some, like ML, have none
- Most, like Java, are somewhere in the middle

Defining coercion in Java

5.6.1 Unary Numeric Promotion
Some operators apply *unary numeric promotion* to a single operand, which must produce a value of a numeric type: If the operand is of compile-time type **`byte`**, **`short`**, or **`char`**, unary numeric promotion promotes it to a value of type **`int`** by a widening conversion (§5.1.2). Otherwise, a unary numeric operand remains as is and is not converted. Unary numeric promotion is performed on expressions in the following situations: the dimension expression in array creations (§15.9); the index expression in array access expressions (§15.12); operands of the unary operators plus **+** (§15.14.3) and minus **-** (§15.14.4) ...

The Java Language Specification
James Gosling, Bill Joy, Guy Steele

Parameter Coercion

if a language supports coercion of parameters on a function call (or of operands when an operator is applied), the resulting function (or operator) is **polymorphic**

Example: Java

```
void f(double x) {
  …
}

f((byte) 1);
f((short) 2);
f('a');
f(3);
f(4L);
f(5.6F);
```

This **f** can be called with any type of parameter Java is willing to coerce to type **double**

f is **polymorphic**

Coercion vs. Overloading

- There are potentially tricky interactions between overloading and coercion
  - Overloading uses the types to choose the definition
  - Coercion uses the definition to choose a type conversion

Ambiguities might appear and each language system has to solve them in some way.

Ambiguity Example

- Suppose that, like C++, a language is willing to coerce **char** to **int** or to **double**

```
int square(int x) {
  return x*x;
}
double square(double x) {
  return x*x;
}
```

- Which **square** gets called for **square('a')** ?

Ambiguity Example

- Suppose that, like C++, a language is willing to coerce **char** to **int**
- Which **f** gets called for **f('a', 'b')** ?

```
void f(int x, char y) {
  …
}
void f(char x, int y) {
  …
}
```

Outline

- Overloading
- Parameter coercion
- **Parametric polymorphism**
- Definitions and classifications

Parametric Polymorphism

- A function exhibits **parametric polymorphism** if it has a type that contains one or more type variables

```
- fun f(a, b) = (a = b);
val f = fn : ''a * ''a -> bool
```

- A type with type variables is a *polytype*
- Found in ML, C++ and Ada, Java

Ex: C++ Function templates

```
// returns the maximum
   of 2 integers

int max (int left, int
   right)
{
if (left < right)
  return right ;
else
  return left ;
}
```

```
//returns the maximum of
   2 doubles

double max (double left,
   double right)
{
if (left < right)
  return right ;
else
  return left ;
}
```

What is the problem here?

What do we need to solve it?

Ex: C++ Function templates

A function template for the function max:

```
template <class T>
T max (T left, T right)
{
if (left < right)
       return right ;
else
      return left ;
}
```

Here “class T” means “type T”. T is a **type variable**

T can be any type for which the operator < is defined

For other types, operator < can be overloaded.

Ex: C++ Function templates

```
#include <iostream>
using namespace std;

.. place here the function template

int main()
{
int integer1 = 4 ;
int integer2 = 10 ;

int max1 = max (integer1, integer2);
cout << "The maximum integer is " << max1 << endl

double double1 = 100.20 ;
double double2 = 5.7 ;

double max2 = max (double1,double2) ;
cout << "The maximum double is " << max2 << endl ;

return 0 ;
}
```

Implementation

- Many copies vs. one copy
- An improved implementation for parametric polymorphism is an active are of programming language research

Outline

- Overloading
- Parameter coercion
- Parametric polymorphism
- Definitions and classifications

So what is polymorphism?

An attempt at a definition

A function or operator is *polymorphic* if it has at least two possible types.

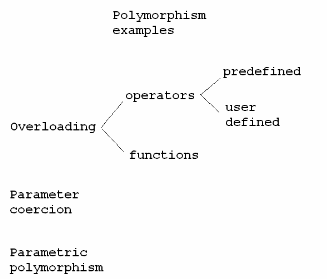
How many types?

- *ad hoc polymorphism* if it has only finitely many possible types
- *universal polymorphism* if it has infinitely many **possible** types

Ad hoc/universal?

- Overloading — Ad hoc
- Parametric coercion — Ad hoc
- Parametric polymorphism — Universal

Summary



Conclusion

- There are many more phenomena that people call polymorphism. We presented only 3 examples and gave a definition that covers them.
- Languages with dynamic type checking do not need polymorphism.
- Polymorphism is a way to gain some freedom and flexibility and still benefit from the static type checking
- Polymorphism is powerful and flexible feature but it presents opportunities for abuse.

Exercises

1. (Weber Ch.8 ex.1) try yourself at home
2. (Weber Ch.8 ex.3) (in class)

Consider an unknown language with integer and real types in which 1+2, 1.0+2, 1+2.0 and 1.0 + 2.0 are all legal expressions.

a. Explain how this could be the result of coercion, using no overloading

b. Explain how this could be the result of overloading using no coercion

c. Explain how this could result from a combination of overloading and coercion

Outline

Part II. Functional programming. A third look at ML

Outline

- More pattern matching
- Function values and anonymous functions
- Higher-order functions and currying
- Some ML predefined higher-order functions

More Pattern-Matching

- Last time we saw pattern-matching in function definitions:

```
fun f 0 = "zero"
 |   f _ = "non-zero";
```

Match Syntax

- A *rule:*

<rule> ::= *<pattern>* **=>** *<expression>*

- A *match* consists of one or more rules separated by a vertical bar, like this:

<match> ::= *<rule>* | *<rule>* '**|**' *<match>*

Case Expressions

```
- case 1+1 of
=    3 => "three" |
=    2 => "two" |
=    _ => "hmm";
val it = "two" : string
```

Example

```
case x of
  _::_::c::_ => c |
  _::b::_ => b |
  a::_ => a |
  nil => 0
```

Generalizes `if`

```
if exp1 then exp2 else exp3
```

```
case exp1 of
  true => exp2 |
  false => exp3
```

- The two expressions above are equivalent
- So **`if`**-**`then`**-**`else`** is really just a special case of **`case`**

Behind the Scenes

- Expressions using **`if`** are actually treated as abbreviations for **`case`** expressions
- This explains some odd SML/NJ error messages:

```
- if 1=1 then 1 else 1.0;
Error: types of rules don't agree [literal]
  earlier rule(s): bool -> int
  this rule: bool -> real
  in rule:
    false => 1.0
```

Outline

- More pattern matching
- **Function values and anonymous functions**
- Higher-order functions and currying
- Predefined higher-order functions

Predefined Functions

- When an ML language system starts, there are many predefined variables
- Some are bound to functions:

```
- ord;
val it = fn : char -> int
- ~;
val it = fn : int -> int
```

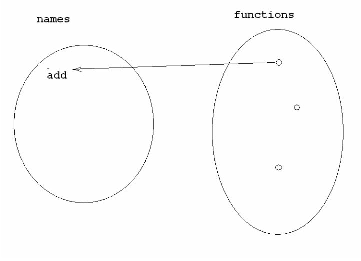
Defining Functions

- We have seen the **`fun`** notation for defining new named functions
- You can also define new names for old functions, using **`val`** just as for other kinds of values:

```
- val x = ~;
val x = fn : int -> int
- x 3;
val it = ~3 : int
```

Function Values

- **Functions in ML *do not have names***
- Just like other kinds of values, function values may be given one or more names by binding them to variables
- The **`fun`** syntax does two separate things:
  - Creates a new function value
  - Binds that function value to a name



Anonymous Functions

- Named function:

fun

```
- fun f x = x + 2;
val f = fn : int -> int
- f 1;
val it = 3 : int
```

- Anonymous function:

fn

```
- fn x => x + 2;
val it = fn : int -> int
- (fn x => x + 2) 1;
val it = 3 : int
```

Using Anonymous Functions

- when you need a small function in just one place and you want to avoid cluttering
- With named function

```
- fun intBefore (a,b) = a < b;
val intBefore = fn : int * int -> bool
- quicksort ([1,4,3,2,5], intBefore);
val it = [1,2,3,4,5] : int list
```

- With anonymous function:

```
- quicksort ([1,4,3,2,5], fn (a,b) => a<b);
val it = [1,2,3,4,5] : int list
- quicksort ([1,4,3,2,5], fn (a,b) => a>b);
val it = [5,4,3,2,1] : int list
```

The **op** keyword

```
- op *;
val it = fn : int * int -> int
- quicksort ([1,4,3,2,5], op <);
val it = [1,2,3,4,5] : int list
```

- Binary operators are special functions
- The keyword **op** before an operator extracts the function used by the operator

Outline

- More pattern matching
- Function values and anonymous functions
- **Higher-order functions and currying**
- Predefined higher-order functions

Higher-order Functions

- Every function has an *order*:
  - A function that does not take any functions as parameters, and does not return a function value, has ***order 1***
  - A function that takes a function as a parameter or returns a function value has ***order n+1***, where *n* is the order of its highest-order parameter or returned value
- The **`quicksort`** we just saw is a second-order function

Practice

What is the order of functions with each of the following ML types?

```
int * int -> bool

int list * (int * int -> bool) -> int list

int -> int -> int

(int -> int) * (int -> int) -> (int -> int)

int -> bool -> real -> string
```

Currying

In ML functions have only one parameter.

Q: How to pass 2 parameters to a function?

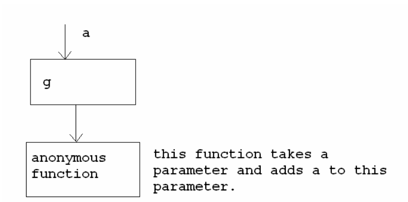
A1: By passing a 2-tuple:

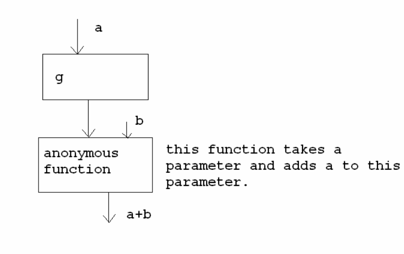
```
fun f (a,b) = a + b;
```

A2: By **currying** = write a function that takes the first argument, and returns another function that takes the second argument and returns the final result:

```
fun g a = fn b => a+b;
```

Haskell B. Curry, (1900-1982)
FP mathematician





Example

```
- fun f (a,b) = a+b;
val f = fn : int * int -> int
- fun g a = fn b => a+b;
val g = fn : int -> int -> int
- f(2,3);
val it = 5 : int
- g 2 3;
val it = 5 : int
```

Remember that function application is left-associative

So `g 2 3` means `((g 2) 3)`

Advantages

- No tuples: we write **`g 2 3`** instead of **`f(2,3)`**
- But the real advantage: we get to specialize functions for particular initial parameters

```
- val add2 = g 2;
val add2 = fn : int -> int
- add2 3;
val it = 5 : int
- add2 10;
val it = 12 : int
```

Advantages

```
- quicksort (op <) [1,4,3,2,5];
val it = [1,2,3,4,5] : int list
- val sortBackward = quicksort (op >);
val sortBackward = fn : int list -> int list
- sortBackward [1,4,3,2,5];
val it = [5,4,3,2,1] : int list
```

Multiple Curried Parameters

- Currying generalizes to any number of parameters

```
- fun f (a,b,c) = a+b+c;
val f = fn : int * int * int -> int
- fun g a = fn b => fn c => a+b+c;
val g = fn : int -> int -> int -> int
- f (1,2,3);
val it = 6 : int
- g 1 2 3;
val it = 6 : int
```

Easier Notation for Currying

- Instead of writing:

  `fun f a = fn b => a+b;`
- We can just write:

  `fun f a b = a+b;`
- This generalizes for any number of curried arguments

```
- fun f a b c d = a+b+c+d;
val f = fn : int -> int -> int -> int -> int
```

Outline

- More pattern matching
- Function values and anonymous functions
- Higher-order functions and currying
- **Predefined higher-order functions**

ML Predefined Higher-Order Functions

- map
- foldr
- foldl

The **map** Function

- Used to apply a function to **every** element of a list, and collect a list of results

```
- map ~ [1,2,3,4];
val it = [~1,~2,~3,~4] : int list
- map (fn x => x+1) [1,2,3,4];
val it = [2,3,4,5] : int list
- map (fn x => x mod 2 = 0) [1,2,3,4];
val it = [false,true,false,true] : bool list
- map (op +) [(1,2),(3,4),(5,6)];
val it = [3,7,11] : int list
```

What is the type of `map`?

The **map** Function Is Curried

```
- map;
val it = fn : ('a -> 'b) -> 'a list -> 'b list
- val f = map (op +);
val f = fn : (int * int) list -> int list
- f [(1,2),(3,4)];
val it = [3,7] : int list
```

Use map function when the result is a list of the same length with the parameter

The **foldr** Function

- Used to combine all the elements of a list (starts from right to left)
- For example, to add up all the elements of a list **x**, we could write **foldr (op +) 0 x**
- It takes a function *f*, a starting value *c*, and a list $x = [x_1, \ldots, x_n]$ and computes:

$$f\left(x_1, f\left(x_2, \cdots f\left(x_{n-1}, f\left(x_n, c\right)\right)\cdots\right)\right)$$

- So **foldr (op +) 0 [1,2,3,4]** evaluates as 1+(2+(3+(4+0)))=10

Foldr: examples

Function start_value list

```
- foldr (op +) 0 [1,2,3,4];
val it = 10 : int
- foldr (op * ) 1 [1,2,3,4];
val it = 24 : int
- foldr (op ^) "" ["abc","def","ghi"];
val it = "abcdefghi" : string
- foldr (op ::) [5] [1,2,3,4];
val it = [1,2,3,4,5] : int list
```

The **foldr** Function Is Curried

```
- foldr;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- foldr (op +);
val it = fn : int -> int list -> int
- foldr (op +) 0;
val it = fn : int list -> int
- val addup = foldr (op +) 0;
val addup = fn : int list -> int
- addup [1,2,3,4,5];
val it = 15 : int
```

The **foldl** Function

- Used to combine all the elements of a list
- Same results as **foldr** in some cases

```
- foldl (op +) 0 [1,2,3,4];
val it = 10 : int
- foldl (op * ) 1 [1,2,3,4];
val it = 24 : int
```

The **foldl** Function

- To add up all the elements of a list **x**, we could write **foldl (op +) 0 x**
- It takes a function f, a starting value c, and a list $x = [x_1, \ldots, x_n]$ and computes:

$$f\left(x_n, f\left(x_{n-1}, \cdots f\left(x_2, f\left(x_1, c\right)\right)\cdots\right)\right)$$

- So **foldl (op +) 0 [1,2,3,4]** evaluates as 4+(3+(2+(1+0)))=10
- Remember, **foldr** did 1+(2+(3+(4+0)))=10

The **foldl** Function

- **foldl** starts at the **l**eft, **foldr** starts at the **r**ight
- Difference does not matter when the function is associative and commutative, like **+** and *****
- For other operations, it does matter

```
- foldr (op ^) "" ["abc","def","ghi"];
val it = "abcdefghi" : string
- foldl (op ^) "" ["abc","def","ghi"];
val it = "ghidefabc" : string
- foldr (op -) 0 [1,2,3,4];
val it = ~2 : int
- foldl (op -) 0 [1,2,3,4];
val it = 2 : int
```

Exercises (Weber, Ch. 9)

- **Exercise 3.** Write a function `squarelist` of type `int list -> int list` that takes a list of integers and returns the list of squares of those integers.

- **Exercise 17.** Write a function `max` of type `int list ->int` that returns the largest element of a list. Your function does not need to behave well if the list is empty.