UNIVERSITETET
I OSLO

Types, Polymorphism and Overloading

Gerardo Schneider
Department of Informatics – University of Oslo

Based on John C. Mitchell's slides

INF 3110/4110 - 2005

Before starting... Some clarifications

- Mandatory exercises must be done **individually**

- Side-effect: a property of a function that modifies some state other than its return value
  - E.g., a function might modify a global variable or one of its arguments; write a result in the screen or in a file.

ML lectures

1. 05.09: A quick introduction to ML
2. 12.09: The Algol Family and more on ML (Mitchell's Chapter 5 + more)
3. **Today: Types, Polymorphism and Overloading (Mitchell's Chapter 6)**
4. 17.10: Exceptions and Continuations (Mitchell's Chapter 8)
5. 24.10: Revision (!?)

Outline

- Types in programming
- Type safety
- Polymorphism*s*
- Type inference
- Type declaration

Type

A type is a collection of computational entities sharing some common property

- Examples
  - Integers
  - [1 .. 100]
  - Strings
  - int → bool
  - (int → int) →bool
- "Non-examples"
  - {3, true, 5.0}
  - Even integers
  - {f:int → int | if x>3 then f(x) > x*(x+1)}

Distinction between types and non-types is language dependent.

Uses for types

- Program organization and documentation
  - Separate types for separate concepts
    - E.g., customer and accounts (banking program)
  - Types can be checked, unlike program comments
- Identify and prevent errors
  - Compile-time or run-time checking can prevent meaningless computations such as 3 + true - “Bill”
- Support optimization
  - Short integers require fewer bits
  - Access record component by known offset

Type errors

- Hardware error
  - Function call x() (where x is not a function) may cause jump to instruction that does not contain a legal op code
- Unintended semantics
  - int_add(3, 4.5): Not a hardware error, since bit pattern of float 4.5 can be interpreted as an integer

General definition of type error

- A *type error* occurs when execution of program is not faithful to the intended semantics
- Type errors depend on the concepts defined in the language; not on ***how*** the program is executed on the underlying software
- All values are stored as sequences of bits
  - Store `4.5` in memory as a floating-point number
    - Location contains a particular bit pattern
  - To interpret bit pattern, we need to know the type
  - If we pass bit pattern to integer addition function, the pattern will be interpreted as an integer pattern
    - Type error if the pattern was intended to represent 4.5

Subtyping

- Subtyping is a relation on types allowing values of one type to be used in place of values of another
  - ***Substitutivity:*** If A is a subtype of B (A<:B), then any expression of type A may be used without type error in any context where B may be used
- In general, if f: A -> B, then f may be applied to x if x: A
  - Type checker: If f: A -> B and x: C, then C = A
- In languages with subtyping
  - Type checker: If f: A -> B and x: C, then C <: A

Remark: No subtypes in ML!

Monomorphism vs. Polymorphism

- *Monomorphic* means ”having only one form”, as opposed to *Polymorphic*
- A type system is monomorphic if each constant, variable, etc. has unique type
- Variables, expressions, functions, etc. are polymorphic if they ”allow” more than one type

Example. In ML, the *identity* function `fn x => x` is polymorphic: it has infinitely many types!

```
- fn x => x
```

Warning! The term ”polymorphism” is used with different specific technical meanings (more on that later)

Outline

- Types in programming
- **Type safety**
- Polymorphism*s*
- Type inference
- Type declaration

Type safety

- A Prog. Lang. is *type safe* if no program can violate its type distinction (e.g. functions and integer)
- Examples of not type safe language features:
  - Type casts (a value of one type used as another type)
    - Use integers as functions (jump to a non-instruction or access memory not allocated to the program)
  - Pointer arithmetic
    - *(p) has type A if p has type A*
    - x = *(p+i) what is the type of x?
  - Explicit deallocation and dangling pointers
    - Allocate a pointer p to an integer, deallocate the memory referenced by p, then later use the value pointed to by p

Relative type-safety of languages

- Not safe: BCPL family, including C and C++
  - Casts; pointer arithmetic
- Almost safe: Algol family, Pascal, Ada.
  - Explicit deallocation; dangling pointers
    - No language with explicit deallocation of memory is fully type-safe
- Safe: Lisp, ML, Smalltalk, Java
  - Lisp, Smalltalk: dynamically typed
  - ML, Java: statically typed

Compile-time vs. run-time checking

- Lisp uses run-time type checking

  (car x)   check first to make sure x is list

- ML uses compile-time type checking

  f(x)   must have f : A → B and x : A

- Basic tradeoff
  - Both prevent type errors
  - Run-time checking slows down execution (compiled ML code, up-to **4** times faster than Lisp code)
  - Compile-time checking restricts program flexibility

    Lisp list: elements can have different types

    ML list: all elements must have same type

Compile-time type checking

- *Sound* type checker: no program with error is considered correct
- *Conservative* type checker: some programs without errors are considered to have errors
- Static typing always conservative

```
if  (possible-infinite-run-expression)
     then  (expression-with-type-error)
     else  (expression-with-type-error)
```

Cannot decide at compile time if run-time error will occur

(from the undecidability of the Turing machine's halting problem)

Outline

- Types in programming
- Type safety
- **Polymorphism*s***
- Type inference
- Type declaration

Polymorphism: three forms

Parametric polymorphism

- Single function may be given (infinitely) many types
- The type expression involves *type variables*

Example: in ML the identity function is polymorphic

```
- fn x => x;
val it = fn : 'a -> 'a
```

This pattern is called *type scheme*

Type variable may be replaced by *any* type

An *instance* of the type scheme may give:

int→int, bool→bool, char→char,
int*string*int→int*string*int, (int→real)→(int→real), ...

Polymorphism: three forms (cont.)

◆ Ad-hoc polymorphism (or Overloading)

- A single symbol has two (or more) meaning (it refers to more than one algorithm)
- Each algorithm may have different type
- Choice of algorithm determined by type context
- Types of symbol may be arbitrarily different

Example: In ML, + has 2 different associated implementations: it can have types int*int→int and real*real→real, no others

Polymorphism: three forms (cont.)

◆ Subtype polymorphism

- The subtype relation allows an expression to have many possible types
- Polymorphism not through type parameters, but through subtyping:
  - If method *m* accept any argument of type *t* then *m* may also be applied to any argument from any subtype of *t*

REMARK 1: In OO, the term "polymorphism" is usually used to denote subtype polymorphism (ex. Java, OCAML, etc)

REMARK 2: ML does not support subtype polymorphism!

Parametric polymorphism

- Explicit: The program contains type variables
  - Often involves explicit instantiation to indicate how type variables are replaced with specific types
  - Example: C++ templates
- Implicit: Programs do not need to contain types
  - The type inference algorithm determines when a function is polymorphic and instantiate the type variables as needed
  - Example: ML polymorphism

Parametric Polymorphism: ML vs. C++

- C++ function template
  - Declaration gives type of funct. arguments and result
  - Place inside template to define type variables
  - Function application: type checker does instantiation
- ML polymorphic function
  - Declaration has no type information
  - Type inference algorithm
    - Produce type expression with variables
    - Substitute for variables as needed

ML also has module system with explicit type parameters

Example: swap two values

- C++

```
void swap (int& x, int& y){
    int tmp=x;  x=y;  y=tmp;
}
```

```
template <typename T>
void swap(T& , T& y){
   T tmp=x; x=y; y=tmp;
}
```

- Instantiations:
  - int i,j; … swap(i,j); //use swap with T replaced with int
  - float a,b;… swap(a,b); //use swap with T replaced with float
  - string s,t;… swap(s,t); //use swap with T replaced with string

Example: swap two values

- ML

```
- fun swap(x,y) =
      let val z = !x in x := !y; y := z end;
val swap = fn : 'a ref * 'a ref -> unit
```

Remark: Declarations look similar in ML and C++,
but compile code is very different!

Parametric Polymorphism: Implementation

- C++
  - Templates are instantiated at program link time
  - Swap template may be stored in one file and the program(s) calling swap in another
  - Linker duplicates code for each type of use
- ML
  - Swap is compiled into one function (no need for different copies!)
  - Typechecker determines how function can be used

Parametric Polymorphism: Implementation

- Why the difference?
  - C++ arguments passed by reference (pointer), but local variables (e.g. tmp, of type T) are on stack
    - Compiled code for swap depends on the size of type T => Need to know the size for proper addressing
  - ML uses pointers in parameter passing (*uniform data representation*)
    - It can access all necessary data in the same way, regardless of its type
- Efficiency
  - C++: more effort at link time and bigger code
  - ML: run more slowly

ML overloading

- Some predefined operators are overloaded
  - + has types int*int→int and real*real→real
- User-defined functions must have unique type
  - fun plus(x,y) = x+y; (compiled to int or real function, not both)

In SML/NJ:

```
- fun plus(x,y) = x+y;
  val plus = fn : int * int -> int
```

If you want to have `plus = fn : real * real -> real` you must provide the type:

```
- fun plus(x:real,y:real) = x+y;
```

ML overloading (cont.)

◆ Why is a unique type needed?

- Need to compile code implies need to know which **+** (different algorithm for distinct types)
- Efficiency of type inference
- Overloading is resolved at compile time
  - Choosing one algorithm among all the possible ones
  - Automatic conversion is possible (**not** in ML!)

Outline

- Types in programming
- Type safety
- Polymorphism*s*
- **Type inference**
- Type declaration

Type checking and type inference

- Type checking: The process of checking whether the types declared by the programmer "agrees" with the language constraints/ requirement
- Type inference: The process of determining the type of an expression based on information given by (some of) its symbols/sub-expressions

ML is designed to make type inference tractable
(one of the reason for not having subtypes in ML!)

Type checking and type inference

- Standard type checking

```
int f(int x) { return x+1; };
int g(int y) { return f(y+1)*2;};
```

  - Look at body of each function and use declared types of identifies to check agreement.

- Type inference

```
~~int~~ f(~~int~~ x) { return x+1; };
~~int~~ g(~~int~~ y) { return f(y+1)*2;};
```

  - Look at code without type information and figure out what types could have been declared.

Type inference algorithm: Some history

- Usually known as Milner-Hindley algorithm
- **1958:** Type inference algorithm given by H.B. Curry and R. Feys for the *typed lambda calculus*
- **1969:** R. Hindley extended the algorithm and proved it gives the most general type
- **1978:** R. Milner -independently of Hindley- provided an equivalent algorithm (for ML)
- **1985:** L. Damas proved its completeness and extended it with polymorphism

ML Type Inference

- Example

  - fun f(x) = 2+x;

  val f = fn : int → int

- How does this work?
  - \+ has two types: int*int → int, real*real→real
  - 2 : int, has only one type
  - This implies + : int*int → int
  - From context, need x: int
  - Therefore f(x:int) = 2+x has type int → int

Overloaded + is unusual. Most ML symbols have unique type.

In many cases, unique type may be polymorphic.

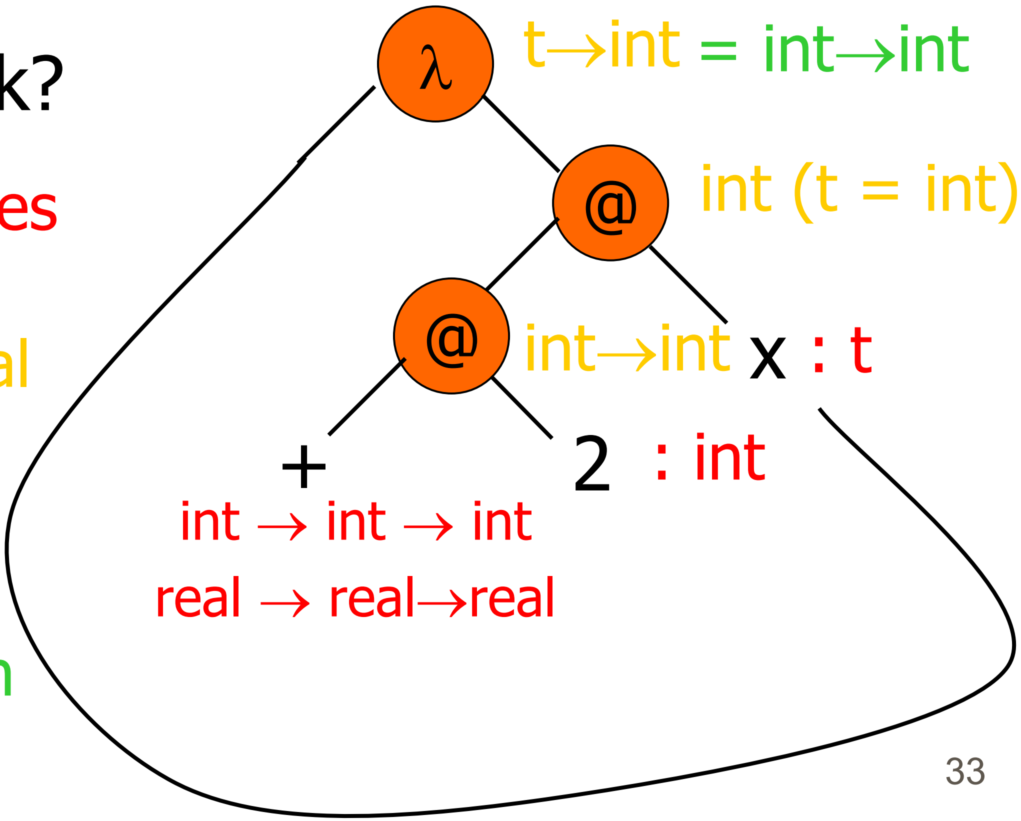
Another presentation

f(x) = 2+x equiv f = λx. (2+x) equiv f = λx. ((plus 2) x)

- ◆ Example
  - fun f(x) = 2+x;

    val f = fn : int → int
- ◆ How does this work?
  1. Assign types to leaves
  2. Propagate to internal nodes and generate constraints
  3. Solve by substitution

Graph for λx. ((plus 2) x)



t→int = int→int

int (t = int)

int→int

x : t

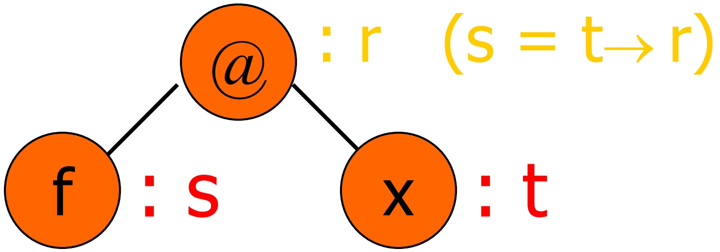
2 : int

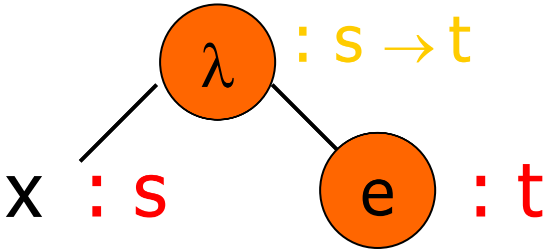
+

int → int → int

real → real→real

Application and Abstraction





- Application
  - f(x)
  - f must have function type domain→ range
  - domain of f must be type of argument x
  - result type is range of f
- Function expression
  - λx.e (fn x => e)
  - Type is function type domain→ range
  - Domain is type of variable x
  - Range is type of function body e

Types with type variables

'a is syntax for "type variable" (t in the graph)

- Example

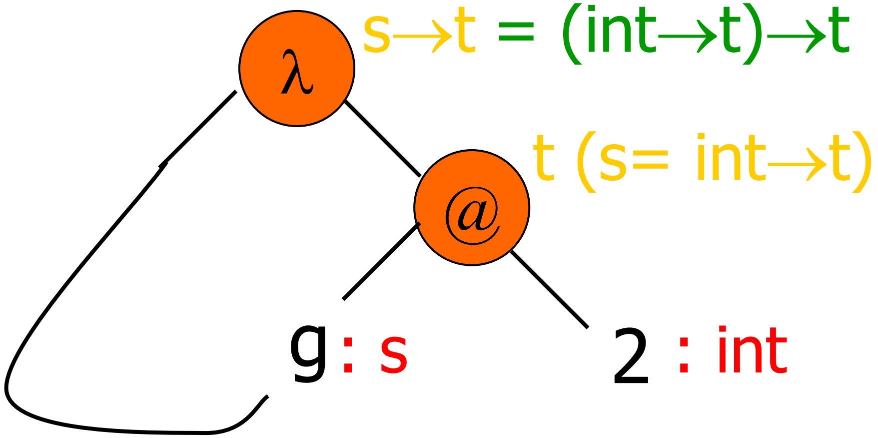
  \- fun f(g) = g(2);

  val f = fn : (int→'a)→'a

- How does this work?

1. Assign types to leaves
2. Propagate to internal nodes and generate constraints
3. Solve by substitution

Graph for λg. (g 2)



Use of Polymorphic Function

- Function

```
- fun f(g) = g(2);
 val f = fn : (int→’a)→’a
```

- Possible applications

g may be the function:

```
- fun add(x) = 2+x;
 val add = fn : int → int
```

Then:

```
- f(add);
 val it = 4 : int
```

g may be the function:

```
- fun isEven(x) = ...;
 val it = fn : int → bool
```

Then:

```
- f(isEven);
  val it = true : bool
```

Recognizing type errors

- Function
  - fun f(g) = g(2);

    val f = fn : (int→'a)→'a
- Incorrect use
  - fun not(x) = if x then false else true;

    val not = fn : bool → bool
  - f(not);

  Why?

  Type error: cannot make bool → bool = int → 'a

Another type inference example

- Function Definition
  - fun f(g,x) = g(g(x));

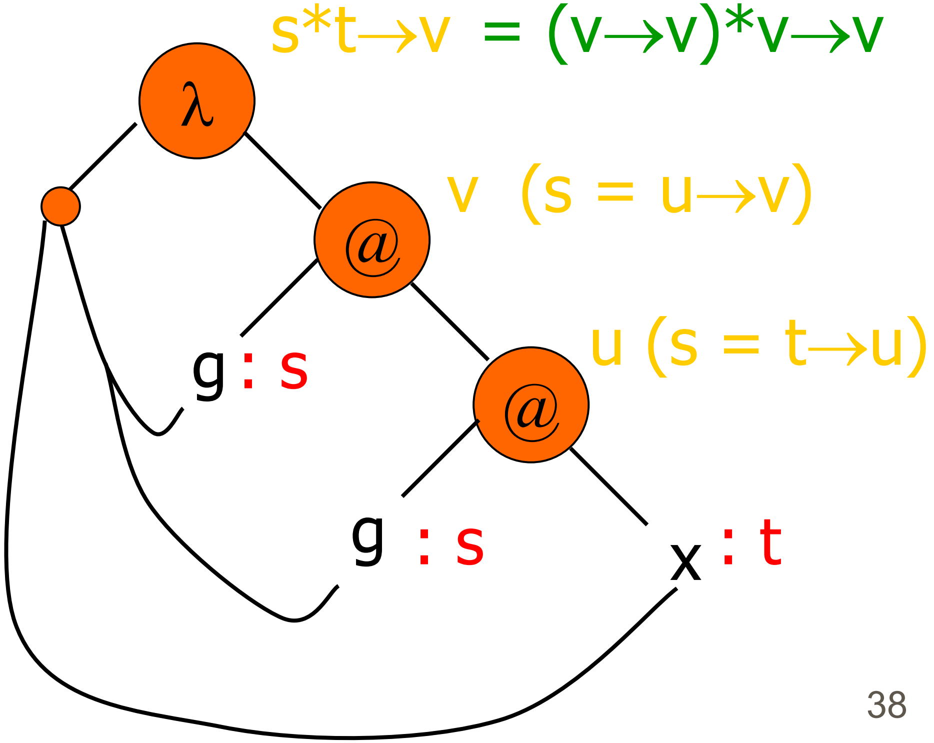
  val f = fn : ('a→'a)*'a → 'a
- Type Inference

  Assign types to leaves

  Propagate to internal nodes and generate constraints

  Solve by substitution

Graph for λ⟨g,x⟩. g(g x)



Polymorphic datatypes

- Datatype with type variable

  ```
  - datatype 'a list = nil | cons of 'a*('a list);
   nil : 'a list
   cons : 'a*('a list) → 'a list
  ```

- Polymorphic function

  ```
  - fun length nil = 0
      |   length (cons(x,rest)) = 1 + length(rest);
    length : 'a list → int
  ```

- Type inference
  - Infer separate type for each clause
  - Combine by making two types equal (if necessary)

Main points about type inference

- Compute type of expression
  - Does not require type declarations for variables
  - Find *most general type* by solving constraints
  - Leads to polymorphism
- Static type checking without type specifications
- May lead to better error detection than ordinary type checking
  - Type may indicate a programming error even if there is no type error (example following slide).

Information from type inference

- An interesting function on lists

```
fun reverse (nil) = nil
|     reverse (x::lst) = reverse(lst);
```

- Most general type

  reverse : 'a list → 'b list

- What does this mean?

  Since reversing a list does not change its type, there must be an error in the definition

  x is not used in "reverse(lst)"!

Outline

- Types in programming
- Type safety
- Polymorphism*s*
- Type inference
- **Type declaration**

Type declaration

- Transparent: alternative name to a type that can be expressed without this name
- Opaque: new type introduced into the program, different to any other

ML has both forms of type declaration

Type declaration: Examples

- Transparent ("type" declaration)

```
- type Celsius = real;
- type Fahrenheit = real;
```

```
- fun toCelsius(x) = ((x-32.0)*0.5556);
 val toCelsius = fn : real → real
```

More information:

```
- fun toCelsius(x: Fahrenheit) = ((x-32.0)*0.5556): Celsius;
  val toCelsius = fn : Fahrenheit → Celsius
```

- Since Fahrenheit and Celsius are synonyms for real, the function may be applied to a real:

```
- toCelsius(60.4);
val it = 15.77904 : Celsius
```

Type declaration: Examples

- Opaque ("datatype" declaration)

```
- datatype A = C of int;
- datatype B = C of int;
```

- A and B are different types
- Since B declaration follows A decl.: C has type int→B

Hence:

```
- fun f(x:A) = x: B;
Error: expression doesn't match constraint [tycon mismatch]
expression: A constraint: B
in expression:  x: B
```

- *Abstract types* are also opaque (Mitchell's chapter 9)

Equality on Types

Two forms of type equality:

- Name type equality: Two type names are equal in type checking only if they are the same name

- Structural type equality: Two type names are equal if the types they name are the same

Example: Celsius and Fahrenheit are structurally equal although their names are different

Remarks – Further reading

- More on subtype polymorphism (Java): Mitchell’s Section 13.3.5

ML lectures

1. 05.09: A quick introduction to ML
2. 12.09: The Algol Family and more on ML (Mitchell’s Chapter 5 + more)
3. Today: Types, Polymorphism and Overloading (Mitchell’s Chapter 6)
4. **17.10: Exceptions and Continuations (Mitchell’s Chapter 8)**
5. 24.10: Revision (!?)