Polymorphism and Type Inference

Gerardo Schneider
gerardo@ifi.uio.no

Department of Informatics – University of Oslo

Based on John C. Mitchell’s slides (Stanford U.)
ML lectures

1. 08.09: The Algol Family and ML (Mitchell’s chap. 5 + more)
2. 15.09: More on ML & types (chap. 5, 6, more)
3. 06.10: More on Types: Type Inference and Polymorphism (chap. 6)
4. 13.10: Control in sequential languages, Exceptions and Continuations (chap. 8)
Outline

◆ Polymorphism

◆ Type inference

◆ Type declaration
Revision - Types

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

◆ Uses for types
  • Program organization and documentation
  • Identify and prevent errors
  • Support optimization

◆ Type safety
  • A Prog. Lang. is **type safe** if no program can violate its type distinction
  • Unsafe elements:
    – Type casts (a value of one type used as another type)
    – Pointer arithmetic
    – Explicit deallocation and dangling pointers

◆ Static/compile-time vs. dynamic/run-time checking
Outline

◆ Polymorphisms
  • parametric polymorphism
  • ad hoc polymorphism
  • subtype polymorphism

◆ 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

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

Example: polymorphic sort

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

- `sort((op<),[1,7,3]);`

> `val it = [1,3,7] : int list`
Polymorphism: three forms (cont.)

◆ Ad-hoc polymorphism (or Overloading)
  • A single symbol has two (or more) meanings (it refers to more than one algorithm)
  • Each algorithm may have different type
  • Overloading is resolved at compile time
  • Choice of algorithm determined by type context

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 declaration inside a template to define type variables
  • Function application: type checker does instantiation automatically

◆ 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++

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

```cpp
void swap(T& x, 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

- val a = ref 3 ; val b = ref 7 ;
> val a = ref 3 : int ref
> val b = ref 7 : int ref
- swap(a,b) ;
> val it = () : unit
- !a ;
> val it = 7 : int

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; Pointers are the same size anyway

Comparison

- C++: more effort at link time and bigger code
- ML: run more slowly, but give smaller code and avoids linking problems
- Global link time errors can be more difficult to find out than local compile errors
ML overloading

◆ Some predefined operators are overloaded
  • + has types \( \text{int}*\text{int}\rightarrow\text{int} \) and \( \text{real}*\text{real}\rightarrow\text{real} \)

◆ 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 \( \text{plus} = \text{fn : real * real -> real} \) 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)
  • Overloading is resolved at compile time
    – The compiler must choose one algorithm among all the
      possible ones
    – Automatic conversion is possible (not in ML!)
    – But in e.g. Java: consider the expression (1 + “foo”);
  • Efficiency of type inference – overloading complicates
    type checking
  • Overloading of user-defined functions is not allowed in
    ML!
  • User-defined overloaded function can be incorporated
    in a fully-typed setting using type classes (Haskell)
Parametric polymorphism vs. overloading

◆ Parametric polymorphism
  • One algorithm for arguments of many different types

◆ Overloading
  • Different algorithms for each type of argument
Outline

- Polymorphisms
- 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
  - Provides a flexible form of compile-time/static type checking

◆ Type inference naturally leads to polymorphism, since the inference uses type variables and some of these might not be resolved in the end

**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

```c
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

```c
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 Robert Feys for the *typed lambda calculus*
- **1969:** Roger Hindley extended the algorithm and proved that it gives the most general type
- **1978:** Robin Milner -independently of Hindley- provided an equivalent algorithm (for ML)
- **1985:** Luis 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
ML Type Inference

◆ Example
   - fun f(g,h) = g(h(0));

◆ How does this work?
   • h must have the type: \( \text{int} \rightarrow \text{'}a \), since 0 is of type \( \text{int} \)
   • this implies that g must have the type: \( \text{'}a \rightarrow \text{'}b \)
   • Then f must have the type:
     \[
     (\text{'}a \rightarrow \text{'}b) \ast (\text{int} \rightarrow \text{'}a) \rightarrow \text{'}b
     \]
The type inference algorithm

◆ Example

- fun f(x) = 2+x;
- (val f = fn x => 2+x ;)
> val f = fn : int → int
Detour: the $\lambda$-calculus

◆ “Entscheidungsproblem”: David Hilbert (1928): Can any mathematical problem be solved (or decided) computationally?

◆ Subproblem: Formalize the notion of decidability or computability

◆ Two formal systems/models:
  • Alonzo Church (1936) - $\lambda$-calculus
  • Alan M. Turing (1936/37) – Turing machines.

◆ $\lambda$-calculus $\rightarrow$ functional programming languages

◆ Turing-machines $\rightarrow$ imperative, sequential programming languages

◆ The models are equally strong (they define the same class of computable functions) (Turing 1936)
Detour: the $\lambda$-calculus

◆ Two ways to construct terms:
  
  • Application: $F A$  
  
  • Abstraction: $\lambda x.e$

  If $e$ is an expression on $x$, then $\lambda x.e$ is a function

  Ex:
  
  $e = 3x+4$.
  
  $\lambda x.e = \lambda x.(3x+4)$

  ( $fn \ x \Rightarrow (3x+4)$ )

  compare with “school book” notation:
  
  if $f(x) = 3x+4$ then $f = \lambda x.(3x+4)$

◆ Rules for computation

  $(\lambda x.(3x+4)) \ 2 \rightarrow (3*2) + 4$
  
  $\lambda x.(3x+4) \rightarrow \lambda y.(3y+4)$  
  
  $(\alpha – \text{conversion})$
  
  $(\lambda x.(3x+4)) \ 2 \rightarrow (3*2) + 4 \rightarrow 10$  
  
  $(\beta – \text{reduction})$
Application and Abstraction

◆ Application \( f \ x \)
  - \( f \) must have function type \( \text{domain} \rightarrow \text{range} \)
  - domain of \( f \) must be type of argument \( x \) (b)
  - the range of \( f \) is the result type (c)
  - thus we know that \( a = b \rightarrow c \)

◆ Abstraction \( \lambda x.e \) (fn \( x \) => e)
  - The type of \( \lambda x.e \) is a function type \( \text{domain} \rightarrow \text{range} \)
  - the domain is the type of the variable \( x \) (a)
  - the range is the type of the function body \( e \) (b)
The type inference algorithm

**Example**
- `fun f(x) = 2+x;`
- `(val f = fn x => 2+x ;)`
  > `val f = fn : int → int`

**How does this work?**

1. **Assign types to expressions**
2. **Generate constraints:**
   - `int→int = u → s`
   - `r = u → s`
3. **Solve by unification/substitution**
Types with type variables

◆ 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

‘a is syntax for “type variable” (t in the graph)
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));

Assign types to leaves

Propagate to internal nodes and generate constraints:
  s = t → u, s = u → v
  t = u, u = v
  t = v

Solve by substitution

Graph for \( \lambda \langle g, x \rangle. g(g(x)) \)
Multiple clause function

- **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 append(nil,l) = l
  - append (x::xs,l) = x:: append(xs,l);
  - val append= fn: 'a list * 'a list → 'a list

- **Type inference**
  - Infer separate type for each clause
    - append: 'a list * 'b -> 'b
    - append: 'a list * 'b -> 'a list
  - Combine by making the two types equal (if necessary) 'b = 'a list
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)”!
Type inference and recursion

- Function definition

  - fun sum(x) = x + sum(x-1);
  > val sum= fn : 'int→'int

  \[
  \text{sum} = \lambda x . ( (+ x) ( \text{sum}( (- x) 1) ) )
  \]
Outline

◆ Polymorphisms
◆ 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

• 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. **08.09:** The Algol Family and ML (Mitchell’s chap. 5 + more)
2. **15.09:** More on ML & types (chap. 5, 6, more)
3. **06.10:** More on Types: Type Inference and Polymorphism (chap. 6)
4. **13.10:** Control in sequential languages, Exceptions and Continuations (chap. 8)