

UNIVERSITETET
I OSLO

Polymorphism and Type Inference

Volker Stolz
stolz@ifi.uio.no

Department of Informatics – University of Oslo

Initially by Gerardo Schneider.
Based on John C. Mitchell's slides (Stanford U.)

IN3040 - 2022

SML lectures

- 14.10. - The Algol Family and SML (Mitchell's chap. 5)
- 21.10. - More on ML & Recursive/Higher-order Functions (chap. 5)
- **4.11. - Types, Type Inference and Polymorphism (chap. 6)**
- 11.11. – Types, continued

IN3040 - 2022

Higher-order functions (functionals)

- filter: apply a predicate to every element of list

```
- fun filter (p, nil) = nil
    | filter (p, (x::xs))  = if p(x) then x :: (filter (p,xs))
                             else filter (p,xs) ;

- val odd = fn : int -> bool
- val mylist = [1,2,3,4,5,6,7,8];
- filter (odd, mylist);                > val it = [1,3,5,7] : int list
- map (fn x => x*x, (filter(odd,mylist)));
                                       > val it = [1,9,25,49] : int list
- val pairs = [(1,2),(4,3),(8,9),(0,9),(0,0),(5,1)] ;
-  filter ((op <) , pairs);
                          > val it = [(1,2),(8,9),(0,9)] : (int * int) list
```

IN3040 - 2022

Curried functions

- A function can have only one argument
  - tuples are used for more than one argument
- Multiple arguments may be realized by giving a function as a result
  - *Currying* -> after the logician Haskell B. Curry
- A function over pairs has type

  'a * 'b -> 'c

  while a curried function has type

  'a -> ('b -> 'c)
- A curried function allows *partial application*: applied to its 1st argument (of type 'a), it results in a function of type 'b -> 'c

IN3040 - 2022

Curried functions

- Example: function to add two numbers
  - fun pluss (x,y) = x + y ;
  > val pluss = fn : int * int -> int
  - pluss (2,3) ;
  ➢ val it = 5 : int

- Curried version of the same function
  - fun cPluss x y = x + y ;
  > val cPluss = fn : int -> int -> int
  - cPluss 2 3 ;
  > val it = 5 : int
  - val addTwo = cPluss 2 ;
  > val addTwo = fn : int -> int
  - addTwo 5 ;
  > val it = 7 : int

Curried functions

- Curry and uncurry

```
- fun curry f x y = f (x,y) ;
> val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c

- fun uncurry f (x,y) = f x y  ;
> val uncurry = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

Example: the map function

- Recall that map can be defined as

```
fun map (f, nil) = nil
   | map (f, x::xs) = f(x) :: map (f,xs);
   > val map = fn : ('a -> 'b) * 'a list -> 'b list

   - map (fn x => x+1, [1,2,3]);
   > val it = [2,3,4] : int list
```

- By currying it, we can define map as

```
fun map f nil = nil
   | map f (x::xs) = (f x) :: map f xs;
  > val map = fn : ('a -> 'b) -> 'a list -> 'b list

  - map (fn x => x+1)  [1,2,3];
     > val it = [2,3,4] : int list
```

More on the map function

- We can have a function having as argument a function which has another function as an argument
- Thanks to currying, we can combine functionals to work on lists of lists

  Example:

  - map (map (fn x => x+1)) [[1], [1,2], [1,2,3]];

  → [map (fn x => x+1) [1], map (fn x => x+1)[1,2], map (fn x => x+1)[1,2,3]]

  → [[2], [2,3], [2,3,4]]

What does it give as a result?

> val it = [[2],[2,3], [2,3,4]] : int list list

Outline

- More recursive examples
- Higher-order functions
- Something about equality
- Types in programming
- Type safety

IN3040 - 2022

Equality

- Equality in (S)ML is defined for many types but not all – E.g., it is defined for:
  - Integers
  - Booleans
  - Strings
  - Characters

- What about floating points (reals), compund types (tuples, records, lists), functions, abstract data types, etc?

Equality

- When are two expressions equal?
  - The so-called *Leibniz's Principle of the Identity of Indiscernables*:

"e1 and e2 are **equal** iff they cannot be distinguished by any operation in the language"

"e1 and e2 are **distinct** iff there is some way to tell them apart"

- What is difficult about Leibniz's Principle?

Problems with Equality

- Equality, as defined by Leibniz's principle, is undecidable

In general, there is no program which determines whether two expressions are equal in Leibniz's sense.

Also:

- Problems with reference cells (aliasing)

```
- val s = ref 1 ; val t = ref 1 ;
- s = t ;                  > false
- !s = !t                  > true
- val s = t ;
- s = t ;                  > true
```

- Polymorphic equality complicates the compiler

Equality Types

- An equality type is a type admiting equality test
- Types admiting equality in (S)ML
  - *int, bool, char, string*
  - *tuples* and *records*, **if all their components admit equality**
  - *datatypes*, **if every constructor's parameter admits equality**
  - *lists* admit equality **if the underlying element type admits equality**
    - Two lists are equal if they have the same length and the same elements in corresponding positions

IN3040 - 2022

Equality Types (cont.)

- Do **not** admit equality in (S)ML
  - *reals*
  - *functions*
  - *tuples, records and datatypes* not mentioned in the previous slide
  - *abstract data types*
- Equality type variable: ”a

```
- fun equals (x,y) = if x = y then true else false ;
> stdIn:7.25 Warning: calling polyEqual
val equals = fn : ''a * ''a -> bool
```

IN3040 - 2022

Equality: Examples

- Equality tests on functions is not computable since
  f = g iff for all x, f(x) = g(x)
- No ”standard” notion of equality for an abstract type
  - What is supposed to be the equality on *trees*? Is it defined structurally? Is it over the list of their elements? By DFS or BFS?
- Ex:

```
fun find x nil = false
  | find x (y :: ys) =   x = y orelse find x ys ;

> = stdIn:30.31 Warning: calling polyEqual
val find = fn : ''a -> ''a list -> bool
```

(don’t worry, only a performance issue)

IN3040 - 2022

Outline

- More recursive examples
- Higher-order functions
- Something about equality
- Types in programming
- Type safety

IN3040 - 2022

Type

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

- Examples
  - Integers
  - [1 .. 100]
  - Strings
  - int $\rightarrow$ bool
  - (int $\rightarrow$ int) $\rightarrow$ bool
- "Non-examples"
  - {3, true, 5.0}
  - Even integers
  - {f:int $\rightarrow$ 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

IN3040 - 2022

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
    - If x = 512, executing x() will jump to location 512 and begin execute "instructions" there
- Unintended semantics
  - int_add(3, 4.5): Not a hardware error, since bit pattern of float 4.5 can be interpreted as an integer

IN3040 - 2022

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

IN3040 - 2022

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!

IN3040 - 2022

Outline

- More recursive examples
- Higher-order functions
- Something about equality
- Types in programming
- Type safety

IN3040 - 2022

Type safety

- A Prog. Lang. is *type safe* if no program can violate its type distinction
  - E.g. use an integer as a function
  - Access memory not allocated to the program.
- 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) (C)
  - 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, Haskell
  - Lisp, Smalltalk: dynamically typed
  - ML, Haskell, Java: statically typed

IN3040 - 2022

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 $\rightarrow$ 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

- Combination of Compile/Run-time eg. Java
  - Static type checking to distinguish arrays and integers
  - Run-time checking to detect array bounds errors

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 is 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)

IN3040 - 2022

Outline

- **Polymorphism*s***
  - **parametric** polymorphism
  - ***ad hoc*** polymorphism
  - **subtype** polymorphism
- Type inference
- Type declaration

IN3040 - 2022

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), ...

IN3040 - 2022

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

IN3040 - 2022

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++

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

```
template <typename T>
 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

IN3040 - 2022

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 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)
  - 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

IN3040 - 2022

Outline

- Polymorphism*s*
- **Type inference**
- Type declaration

IN3040 - 2022

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

```
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 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

IN3040 - 2022

ML Type Inference

- Example
  - fun f(g,h) = g(h(0));

- How does this work?
  - h must have the type: int → ´a, since 0 is of type int
  - this implies that g must have the type: ´a → ´b
  - Then f must have the type:

    (´a → ´b) * (int → ´a) → ´b

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)”!

The type inference algorithm

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

- Example

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

IN3040 - 2022

Detour: the λ-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) - λ-calculus
  - Alan M. Turing (1936/37) – Turing machine
- λ-calculus → functional programming languages
- Turing-machines → imperative, sequential programming languages
- The models are equally strong (they define the same class of computable functions) (Turing 1936)

Detour: the λ-calculus

- Two ways to construct terms:
  - Application: F A (or F(A))
  - Abstraction: λx.e

    If e is an expression on x, then λx.e is a function
    Ex:

    e = 3x+4 .

    λx.e = λx.(3x+4) (fn x => (3x+4))

    compare with "school book" notation:

    if f(x) = 3x+4 then f = λx.(3x+4)
- Rules for computation

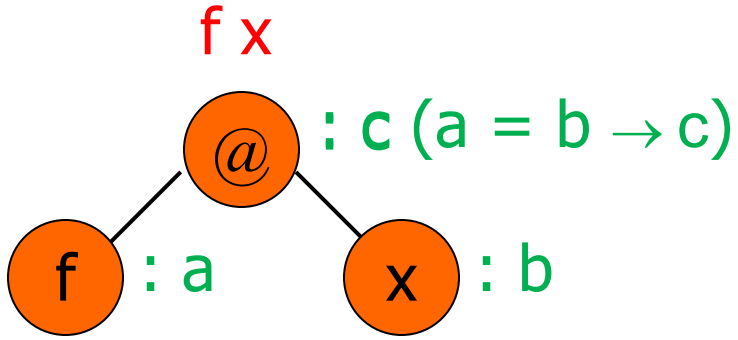
  (λx.(3x+4)) 2 → (3*2) + 4

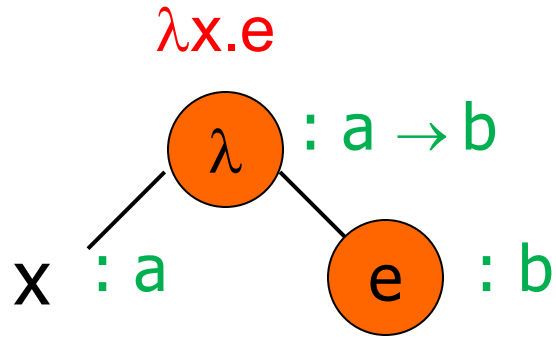
  λx.(3x+4) → λy.(3y+4) (α – conversion)

  (λx.(3x+4)) 2 → (3*2) + 4 → 10 (β – reduction)

IN3040 - 2022

Application and Abstraction





- Application f x
  - f must have function type domain→ 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→ c
- Abstraction λx.e (fn x => e)
  - The type of λx.e is a function type domain→ range
  - the domain is the type of the variable x (a)
  - the range is the type of the function body e (b)

IN3040 - 2022

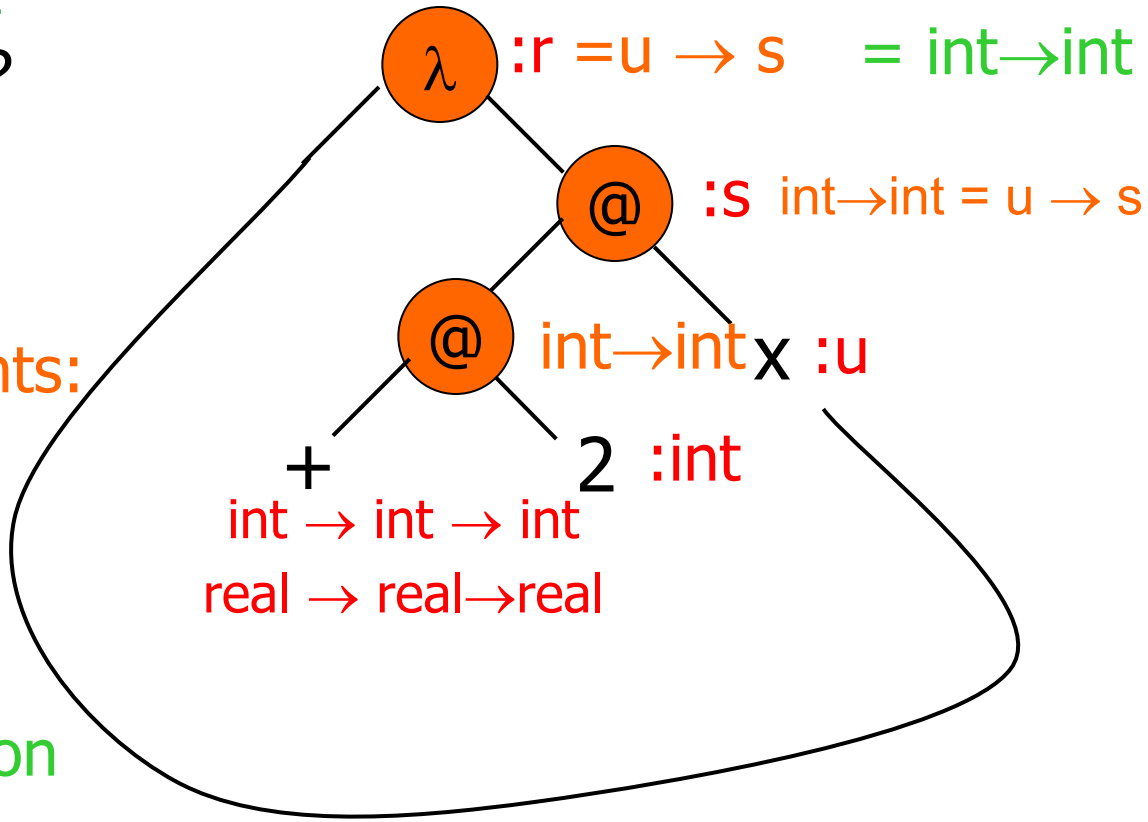
The type inference algorithm

- 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

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

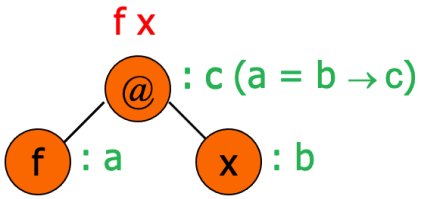
Graph for λx. ((+ 2) x)

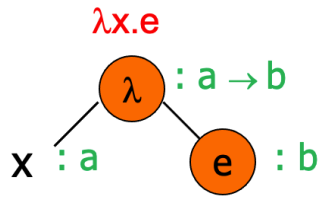


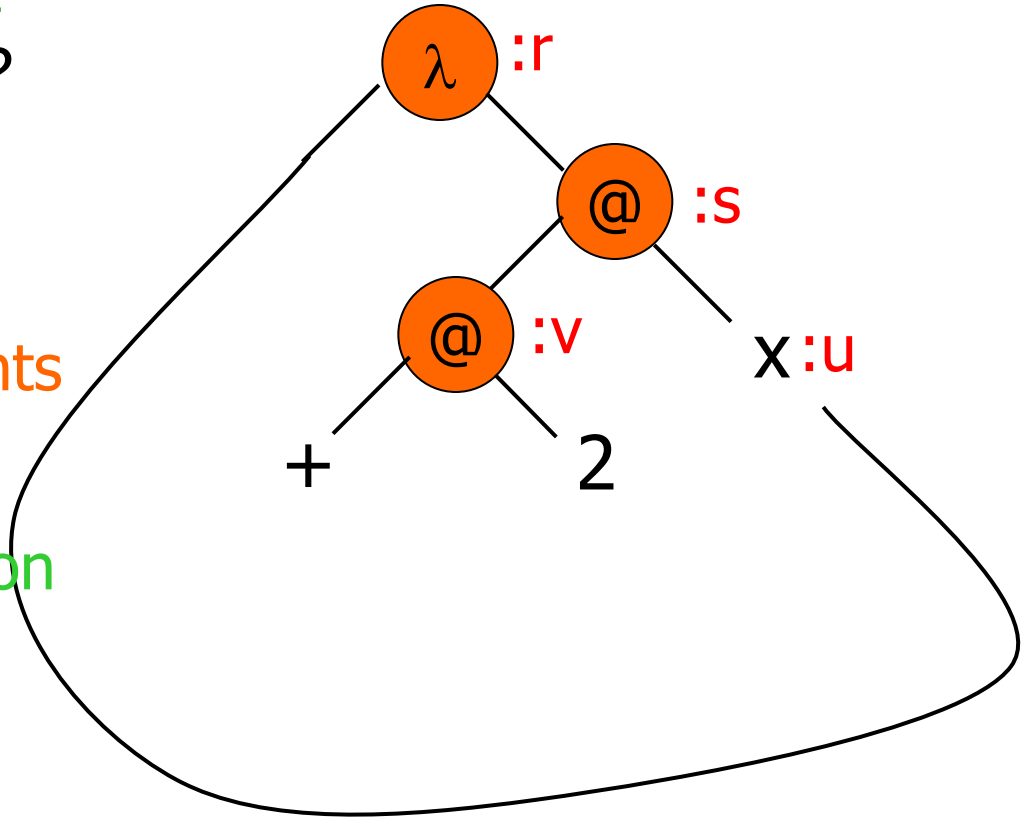
- Example
  - fun f(x) = 2+x;
  - (val f = fn x => 2+x ;)

  > val f = fn : int → int
- How does this work?

1. Assign types to expressions
2. Generate constraints
3. Solve by unification/substitution





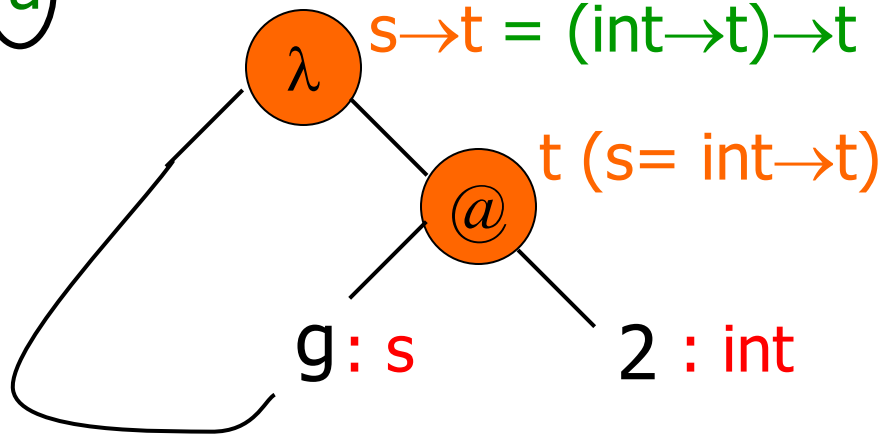


Types with type variables

- Example
  - fun f(g) = g(2);
  - > val f = fn : (int→'a)→'a
- How does this work?

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

Graph for λg. (g 2)



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

IN3040 - 2022

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);

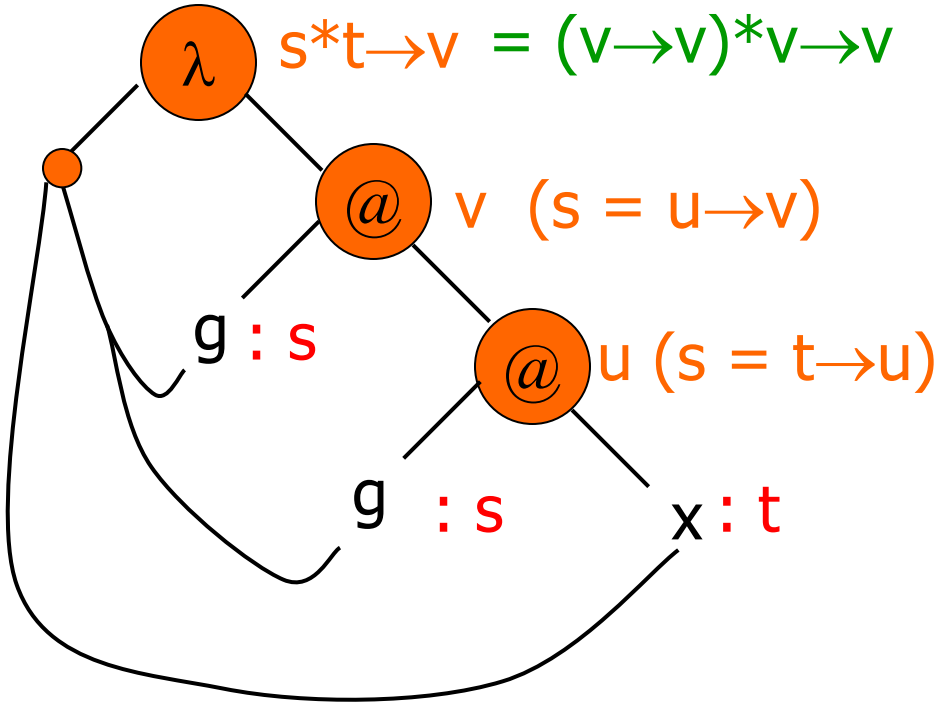
  Why?

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

Another type inference example

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

Graph for λ⟨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

IN3040 - 2022

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).

Type inference and recursion

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

  sum = λx .((+ x) (sum((- x) 1)))

Outline

- 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

IN3040 - 2022

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
```

IN3040 - 2022

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
```

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

IN3040 - 2022