Types

John Mitchell

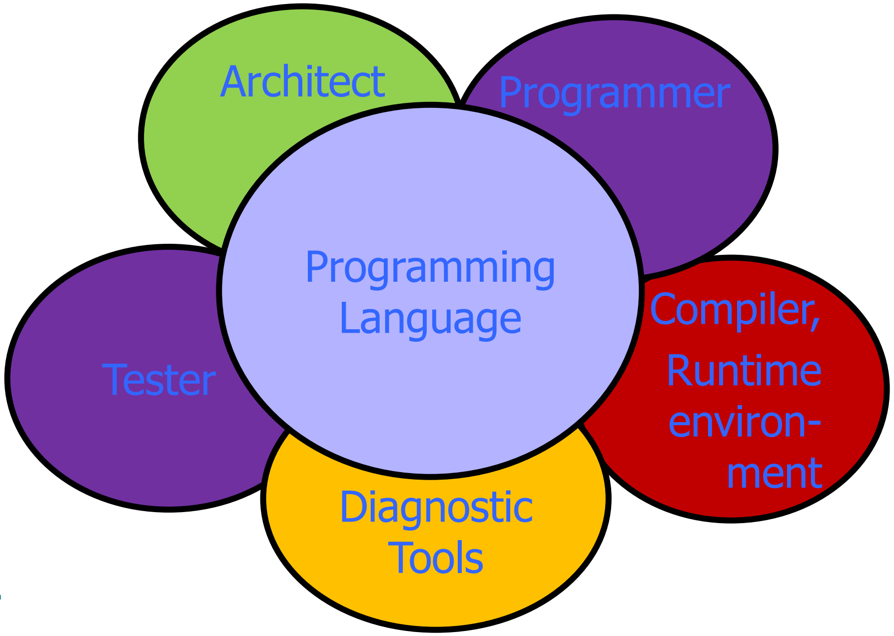
Reading: Chapter 6

Outline

- General discussion of types
  - What is a type?
  - Compile-time vs run-time checking
  - Conservative program analysis
- Type inference
  - Good example of static analysis algorithm
  - Will study algorithm and examples
- Polymorphism
  - Polymorphism vs overloading
  - Uniform vs non-uniform impl of polymorphism

Language goals and trade-offs

- Thoughts to keep in mind
  - What features are convenient for programmer?
  - What other features do they prevent?
  - What are design tradeoffs?
    - Easy to write but harder to read?
    - Easy to write but poorer error messages?
  - What are the implementation costs?



Type

A type is a collection of computable values that share some structural property.

- Examples
  - Integers
  - Strings
  - int $\rightarrow$ bool
  - (int $\rightarrow$ int) $\rightarrow$bool
- "Non-examples"
  - $\{3, \text{true}, \lambda x.x\}$
  - Even integers
  - $\{f{:}\text{int} \rightarrow \text{int} \mid \text{if } x>3 \text{ then } f(x) > x*(x+1)\}$

Distinction between sets that are types and sets that are not types is language dependent.

Uses for types

- Program organization and documentation
  - Separate types for separate concepts
    - Represent concepts from problem domain
  - Indicate intended use of declared identifiers
    - 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
  - Example: short integers require fewer bits
  - Access record component by known offset

Compile-time vs run-time checking

- JavaScript, Lisp use run-time type checking

  f(x)   make sure f is a function *before* calling f

```
js> var f= 3;
js> f(2);
typein:3: TypeError: f is not a function
js>
```

- 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
  - Compile-time checking restricts program flexibility

    JavaScript array: elements can have different types

    ML list: all elements must have same type
  - Which gives better programmer diagnostics?

Expressiveness

- In JavaScript, we can write function like

  ```
  function f(x) { return x < 10 ? x : x(); }
  ```

  Some uses will produce type error, some will not

- Static typing always conservative

  ```
  if  (big-hairy-boolean-expression)
        then  f(5);
        else   f(10);
  ```

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

Relative type-safety of languages

- Not safe: BCPL family, including C and C++
  - Casts, pointer arithmetic
- Almost safe: Algol family, Pascal, Ada.
  - Dangling pointers.
    - Allocate a pointer p to an integer, deallocate the memory referenced by p, then later use the value pointed to by p
    - No language with explicit deallocation of memory is fully type-safe
- Safe: Lisp, ML, Smalltalk, JavaScript, and Java
  - Lisp, Smalltalk, JavaScript: dynamically typed
  - ML, Java: statically typed

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.

ML is designed to make type inference tractable.

Motivation

- Types and type checking
  - Type systems have improved steadily since Algol 60
  - Important for modularity, compilation, reliability

- Type inference
  - Widely regarded as important language innovation
  - ML type inference is an illustrative example flow-insensitive static analysis algorithm

ML Type Inference

- Example
  - fun f(x) = 2+x;

  > val it = 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

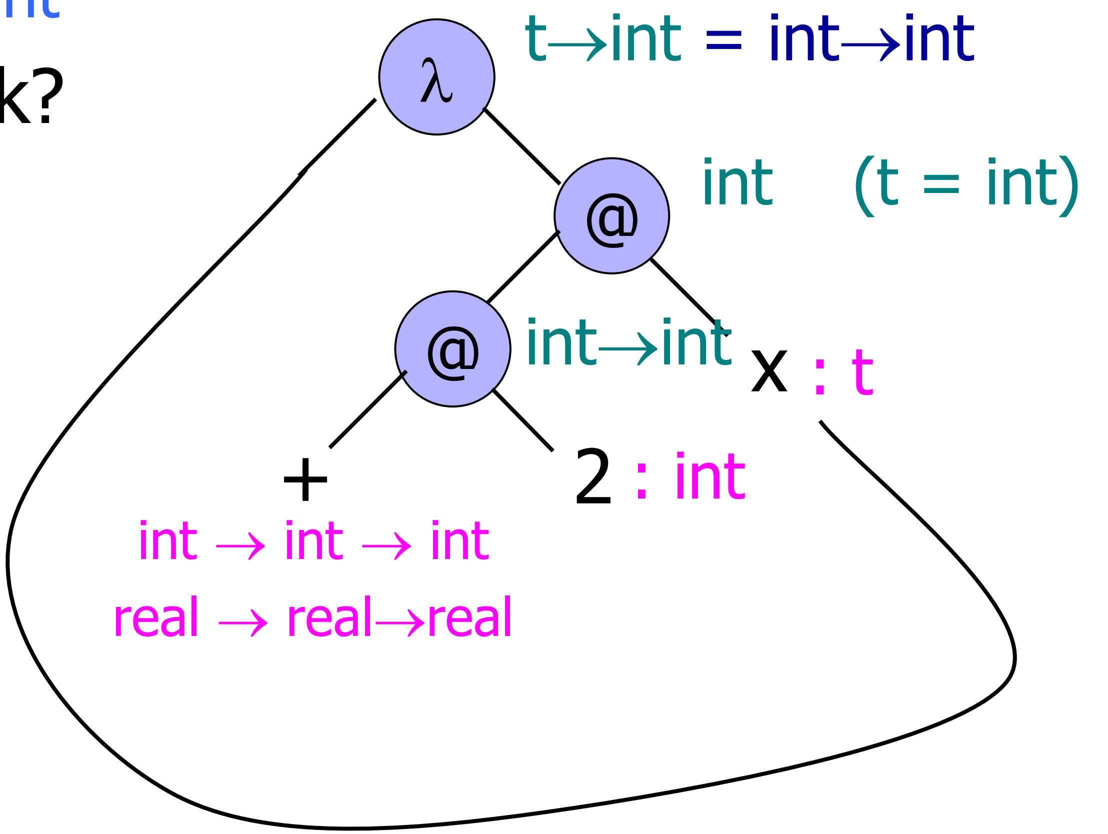
- Example
  - fun f(x) = 2+x;
  - > val it = fn : int → int
- How does this work?

Assign types to leaves

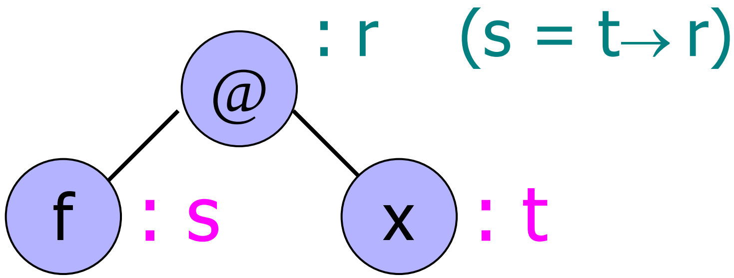
Propagate to internal nodes and generate constraints

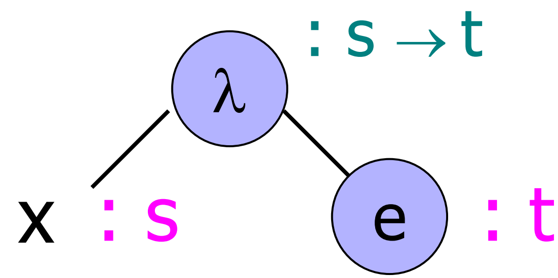
Solve by substitution

Graph for λx. ((plus 2) x)



Application and Abstraction





- Application
  - f must have function type domain→ range
  - domain of f must be type of argument x
  - result type is range of f
- Function expression
  - Type is function type domain→ range
  - Domain is type of variable x
  - Range is type of function body e

Types with type variables

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

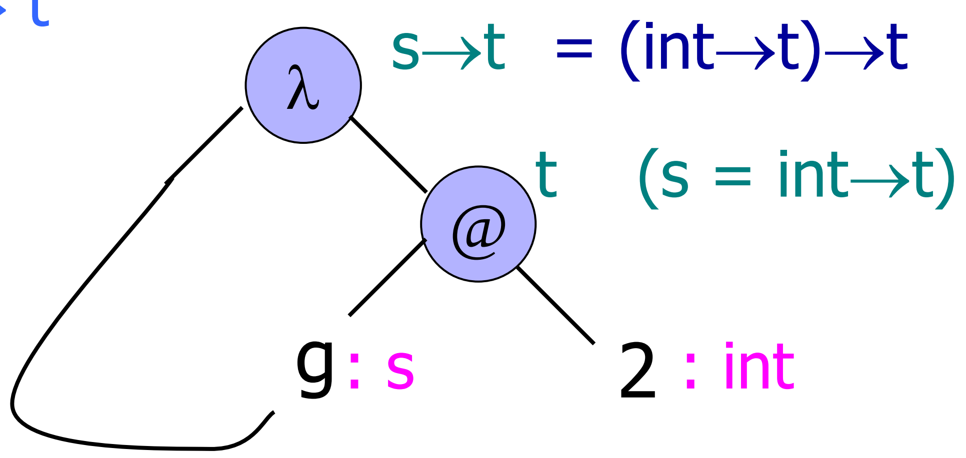
  > val it = fn : (int → t) → t
- How does this work?

Assign types to leaves

Propagate to internal nodes and generate constraints

Solve by substitution

Graph for λg. (g 2)



Use of Polymorphic Function

- Function
  - fun f(g) = g(2);
  > val it = fn : (int → t) → t
- Possible applications
  - fun add(x) = 2+x;
  > val it = fn : int → int
  - f(add);
  > val it = 4 : int

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

Recognizing type errors

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

  > val it = fn : (int $\rightarrow$ t) $\rightarrow$ t
- Incorrect use

  \- fun not(x) = if x then false else true;

  > val it = fn : bool $\rightarrow$ bool

  \- f(not);

  Type error: cannot make bool $\rightarrow$ bool = int $\rightarrow$ t

Another Type Inference Example

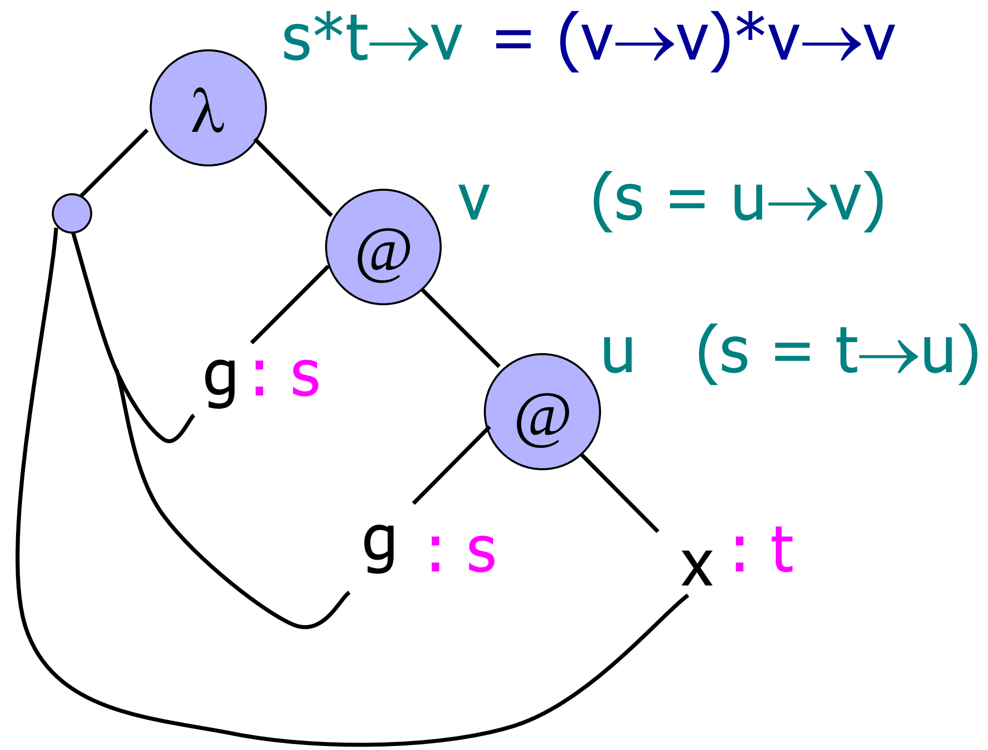
- Function Definition
  - fun f(g,x) = g(g(x));
  - > val it = fn : (t → t)*t → t
- Type Inference

  Assign types to leaves

  Propagate to internal nodes and generate constraints

  Solve by substitution

Graph for λ⟨g,x⟩. g(g x)



Polymorphic Datatypes

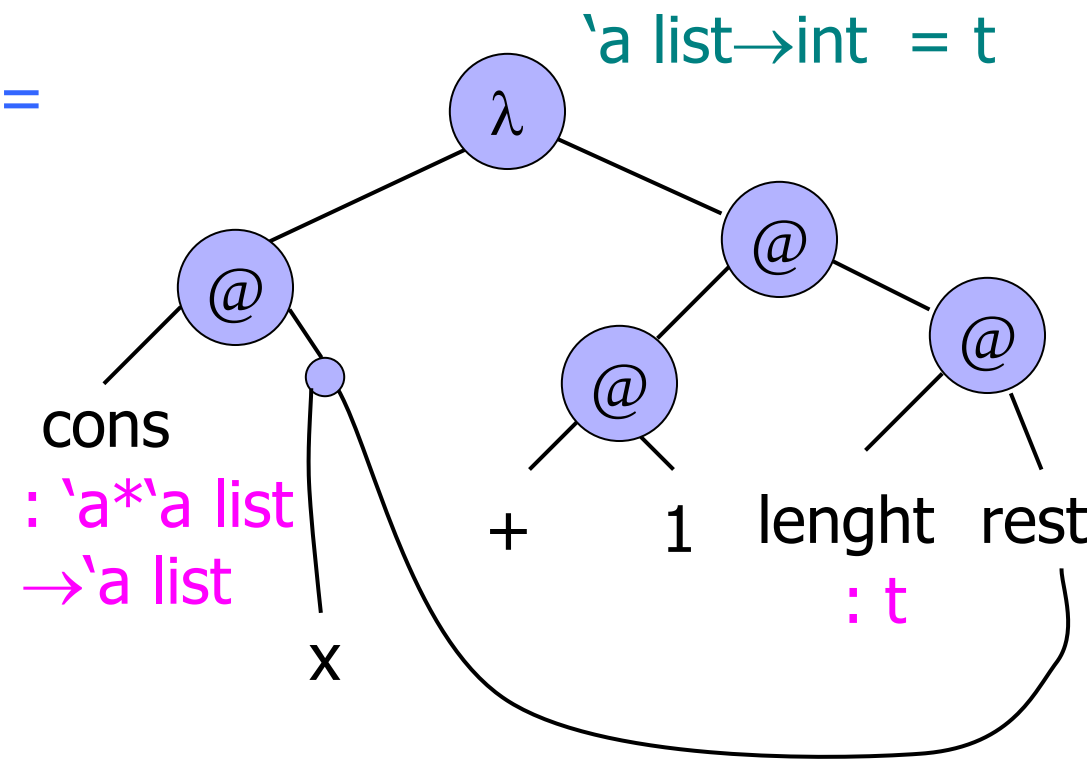
- Datatype with type variable 'a is syntax for "type variable a"
  - datatype 'a list = nil | cons of 'a*('a list)
  - \> nil : 'a list
  - \> cons : 'a*('a list) → 'a list
- Polymorphic function
  - fun length nil = 0
    | length (cons(x,rest)) = 1 + length(rest)
  - \> length : 'a list → int
- Type inference
  - Infer separate type for each clause
  - Combine by making two types equal (if necessary)

Type inference with recursion

- Second Clause

  length(cons(x,rest)) =
    1 + length(rest)

- Type inference
  - Assign types to leaves, including function name
  - Proceed as usual
  - Add constraint that type of function body = type of function name



We do not expect you to master this.

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
- Sometimes better error detection than type checking
  - Type may indicate a programming error even if there is no type error (example following slide)
- Some costs
  - More difficult to identify program line that causes error
  - ML requires different syntax for integer 3, real 3.0

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 of “reverse”

Polymorphism vs Overloading

- Parametric polymorphism
  - Single algorithm may be given many types
  - Type variable may be replaced by *any* type
  - $f : t \to t$ => $f : int \to int$, $f : bool \to bool$, ...
- Overloading
  - A single symbol may refer 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
  - + has types $int*int \to int$, $real*real \to real$, *no others*

Parametric Polymorphism: ML vs C++

- ML polymorphic function
  - Declaration has no type information
  - Type inference: type expression with variables
  - Type inference: substitute for variables as needed
- C++ function template
  - Declaration gives type of function arg, result
  - Place inside template to define type variables
  - Function application: type checker does instantiation

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
```

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

Declarations look similar, but compiled very differently

Implementation

- ML
  - Swap is compiled into one function
  - Typechecker determines how function can be used
- C++
  - Swap is compiled into linkable format
  - Linker duplicates code for each type of use
- Why the difference?
  - ML ref cell is passed by pointer, local x is pointer to value on heap
  - C++ arguments passed by reference (pointer), but local x is on stack, size depends on type

Another example

- C++ polymorphic sort function

```
template <typename T>
void sort( int count, T * A[count] ) {
   for (int i=0; i<count-1; i++)
       for (int j=i+1; j<count-1; j++)
           if (A[j] < A[i]) swap(A[i],A[j]);
}
```

- What parts of implementation depend on type?
  - Indexing into array
  - Meaning and implementation of <

ML Overloading

- Some predefined operators are overloaded
- User-defined functions must have unique type

  - fun plus(x,y) = x+y;

  This is compiled to int or real function, not both
- Why is a unique type needed?
  - Need to compile code $\Rightarrow$ need to know which +
  - Efficiency of type inference

Summary

- Types are important in modern languages
  - Program organization and documentation
  - Prevent program errors
  - Provide important information to compiler
- Type inference
  - Determine best type for an expression, based on known information about symbols in the expression
- Polymorphism
  - Single algorithm (function) can have many types
- Overloading
  - Symbol with multiple meanings, resolved at compile time