Types and Parametric Polymorphism

Vitaly Shmatikov

Reading Assignment

- Mitchell, Chapter 6
- C Reference Manual, Chapters 5 and 6

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

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

Operations on Typed Values

- Often a type has operations defined on values of this type
  - Integers: + - / * < > ...   Booleans: $\wedge$ $\vee$ $\neg$ ...
- Set of values is usually finite due to internal binary representation inside computer
  - 32-bit integers in C: –2147483648 to 2147483647
  - Addition and subtraction may overflow the finite range, so sometimes **`a + (b + c)`** $\neq$ **`(a + b) + c`**
  - Exceptions: unbounded fractions in Smalltalk, unbounded Integer type in Haskell
  - Floating point problems

Type Errors

- Machine data carries no type information
  - 01000000010110000000000000000000 means...
  - Floating point value 3.375? 32-bit integer 1,079,508,992? Two 16-bit integers 16472 and 0? Four ASCII characters @ X NUL NUL?
- A type error is any error that arises because an operation is attempted on a value of a data type for which this operation is undefined
  - Historical note: in Fortran and Algol, all of the types were built in. If needed a type “color,” could use integers, but what does it mean to multiply two colors?

Static vs. Dynamic Typing

- Type system imposes constraints on use of values
  - Example: only numeric values can be used in addition
  - Cannot be expressed syntactically in EBNF
- Language can use static typing
  - Types of all variables are fixed at compile time
  - Example?
- ... or dynamic typing
  - Type of variable can vary at run time depending on value assigned to this variable
  - Example?

Strong vs. Weak Typing

- A language is strongly typed if its type system allows all type errors in a program to be detected either at compile time or at run time
  - A strongly typed language can be either statically or dynamically typed!
- Union types are a hole in the type system of many languages (why?)
- Most dynamically typed languages associate a type with each value

Compile- vs. Run-Time Checking

- Type-checking can be done at compile time
  - Examples: C, ML

f(x) must have f : A $\rightarrow$ B and x : A

- ... or run time

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

  - Examples: Perl, JavaScript
- Java does both
- Basic tradeoffs
  - 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 vs. Safety

- 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, so can’t define the above function

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

Enumeration Types

- User-defined set of values
  - enum day {Monday, Tuesday, Wednesday,
    Thursday, Friday, Saturday, Sunday};

    enum day myDay = Wednesday;
  - In C/C++, values of enumeration types are represented as integers: 0, ..., 6
- More powerful in Java:
  - ```
    for (day d : day.values())
        System.out.println(d);
    ```
    ```

Pointers

- C, C++, Ada, Pascal
- Value is a memory address
  - Remember r-values and l-values?
- Allows indirect referencing
- Pointers in C/C++
  - If T is a type and ref T is a pointer:
    $\& : T \rightarrow \text{ref } T \qquad * : \text{ref } T \rightarrow T \qquad *(\&x) = x$
- Explicit access to memory via pointers can result in erroneous code and security vulnerabilities

Arrays

- Example: float x[3][5];
- Indexing []
  - Type signature: T[ ] x int → T
  - In the above example, type of x: float[ ][ ],
    type of x[1]: float[ ], type of x[1][2]: float
- Equivalence between arrays and pointers
  - a = &a[0]
  - If either e1 or e2 is type: ref T,
    then e1[e2] = *((e1) + (e2))
  - Example: a is float[ ] and i int, so a[i] = *(a + i)

Strings

- Now so fundamental, directly supported by languages
- C: a string is a one-dimensional character array terminated by a NULL character (value = 0)
- Java, Perl, Python: a string variable can hold an unbounded number of characters
- Libraries of string operations and functions
  - Standard C string libraries are unsafe!

Structures

- Collection of elements of different types
  - Not in Fortran, Algol 60, used first in Cobol, PL/I
  - Common to Pascal-like, C-like languages
  - Omitted from Java as redundant

```
struct employeeType {
    char name[25];
    int age;
    float salary;
};
struct employeeType employee;
...
employee.age = 45;
```

Unions

- union in C, case-variant record in Pascal
- Idea: multiple views of same storage

```
type union =
        record
                case b : boolean of
                        true : (i : integer);
                        false : (r : real);
        end;
var  tagged : union;
begin tagged := (b => false, r => 3.375);
      put(tagged.i);  -- error
```

Recursive Datatypes

- data Value = IntValue Integer | FloatValue Float |
  BoolValue Bool | CharValue Char
  deriving (Eq, Ord, Show)
- data Expression = Var Variable | Lit Value |
  Binary Op Expression Expression |
  Unary Op Expression
  deriving (Eq, Ord, Show)
- type Variable = String
- type Op = String
- type State = [(Variable, Value)]

Functions as Types

- Pascal example:

  function newton(a, b: real; function f: real): real;

  - Declares that f returns a real value, but the arguments to f are unspecified

- Java example:

  public interface RootSolvable {double valueAt(double x);}

  public double Newton(double a, double b, RootSolvable f);

Type Equivalence

- Pascal Report:

  "The assignment statement serves to replace the current value of a variable with a new value specified as an expression … The variable (or the function) and the expression must be of identical type"

- Nowhere does it define identical type
  - Which of the following types are equivalent?

```
struct complex { float re, im; };
struct polar { float x, y; };
struct { float re, im; } a, b;
struct complex c,d;   struct polar e;   int f[5], g[10];
```

Subtypes

- A subtype is a type that has certain constraints placed on its values or operations
- Can be directly specified in some languages (Ada)

  subtype one_to_ten is Integer range 1 .. 10;

- Will talk more about subtyping when talking about object-oriented programming

Overloading

- An operator or function is overloaded when its meaning varies depending on the types of its operands or arguments or result
- Examples:
  - Addition: integers and floating-point values
    - Can be mixed: one operand an int, the other floating point
    - Also string concatenation in Java
  - Class PrintStream in Java:

    print, println defined for boolean, char, int, long, float, double, char[ ], String, Object

Function Overloading in C++

- Functions that have the same name but can take arguments of different types

```
inline void swap(int& a, int& b) { int temp = a; a = b; b = temp; }
inline void swap(char& a, char&b) { char temp = a; a = b; b = temp; }
inline void swap(float& a, float& b) { float temp = a; a =b; b = temp }
```

Tells compiler (<u>not</u> preprocessor) to substitute the code of the function at the point of invocation

- Saves the overhead of a procedure call
- Preserves scope and type rules as if a function call was made

Overloading Infix Operators in C++

```
class Complex {
private:
     long double r; // real part
     long double i; // imaginary part
public:
     /* "Complex object constructor function" */
     Complex () { r = 0.0; i = 0.0; }
     Complex (double real, double imag) { r = real; i = imag; }
     ...
     /* "friend" functions can access the private data of a Complex object */
     friend Complex operator+ (Complex a, Complex b) { return Complex(a.r+b.r, a.i+b.i); }
     friend Complex operator- (Complex a, Complex b) { return Complex(a.r-b.r, a.i-b.i); }
     friend Complex operator* (Complex a, Complex b) { return ...; }
     friend Complex operator/ (Complex a, Complex b) { return ...; }
};
```

```
Complex x; // same as Complex x(0.0,0.0);
Complex a(1.0, 0.0);
Complex b(2.5, 3.0);
Complex c(2.0, 2.0);
...
Complex r = a + b * c;  // a + (b * c) --- you can't change associativity in C++
```

Cannot change position, associativity or precedence

Operator Overloading in ML

- ML infers which function to use from the type of the operands

```
- 3 + 5;
val it = 8 : int
- 3.14 + 2.0;
val it = 5.14 : real
- 3.14 + 2;
stdIn:1.1-2.4 Error: operator and operand don't agree [literal]
  operator domain: real * real
  operand:         real * int
  in expression:
    + : overloaded (3.14,(2 : int))
```

User-Defined Infix Operators in ML

```
- infix xor;
infix xor

- fun p xor q = (p orelse q) andalso not (p andalso q);
val xor = fn : bool * bool -> bool

- true xor false xor true;
val it = false : bool
```

- Precedence is specified by integer values 0-9
  - 0 = lowest precedence; left associativity (or else use infixr)
  - nonfix turns infix function into a binary prefix function

```
- infix 6 plus;
infix 6 plus
- fun a plus b = "(" ^ a ^ "+" ^ b ^ ")";
val plus = fn : string * string -> string

- infix 7 times;
infix 7 times
- fun a times b = "(" ^ a ^ "*" ^ b ^ ")";
val times = fn : string * string -> string
```

Polymorphism and Generics

- An operator or function is polymorphic if it can be applied to any one of several related types
  - Enables code re-use!
- Example: generic functions in C++
  - Function operates in exactly the same way regardless of the type of its arguments

```
template<class type> void swap(type& a, type& b) { type temp = a; a = b; b = temp; }
```

  - For each use, compiler substitutes the actual type of the arguments for the 'type' template parameters

```
void swap(int& a, int& b) { int temp = a; a = b; b = temp; }
```

  - This is an example of parametric polymorphism

Polymorphism vs. Overloading

Do you see the difference?

- Parametric polymorphism
  - Single algorithm may be given many types
  - Type variable may be replaced by any type
  - $f : t \to t \;\Rightarrow\; f : int \to int,\ f : bool \to bool, \ldots$
- 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 $int*int \to int$, $real*real \to real$

Type Checking vs. Type Inference

- Standard type checking

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

  - Look at the body of each function and use declared types of identifiers to check agreement
- Type inference

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

  - Look at the 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 of a flow-insensitive static analysis algorithm
    - What does this mean?

ML Type Inference

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

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

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

In many cases, unique type may be polymorphic.

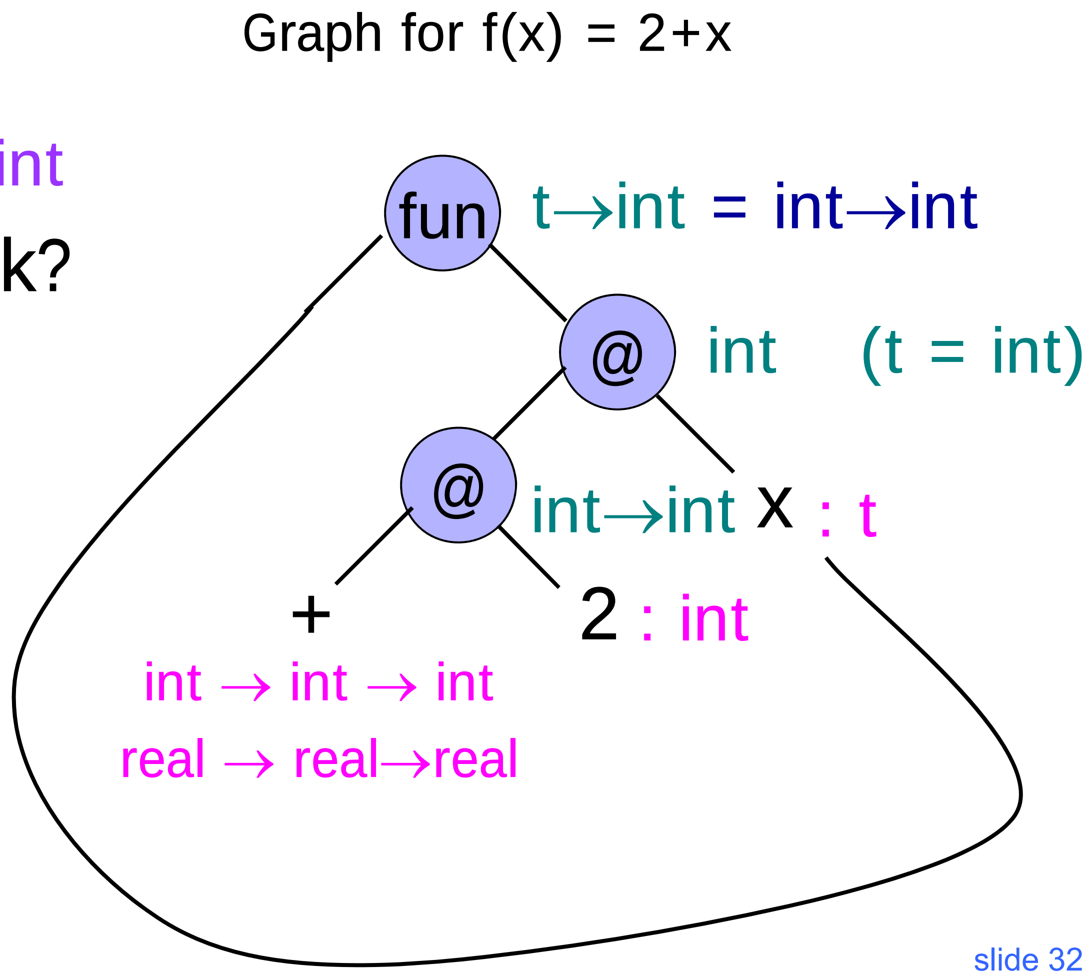
How Does This Work?

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

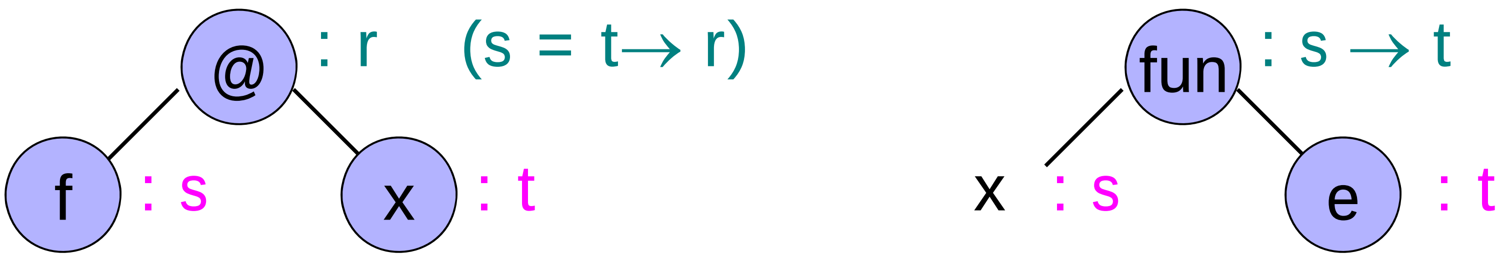
Assign types to leaves

Propagate to internal nodes and generate constraints

Solve by substitution



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 the type of function body e

Types with Type Variables

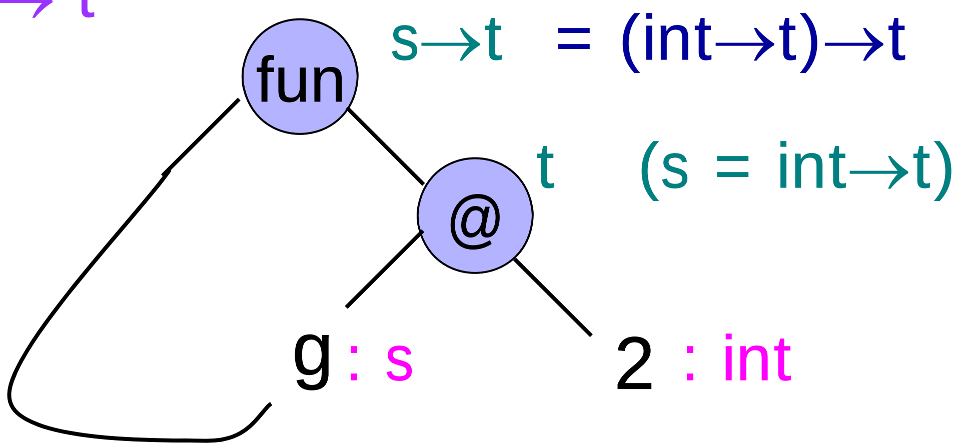
- Example
  - fun f(g) = g(2);
  - > val it = fn : (int → t) → t
- How does this work?

Assign types to leaves

Propagate to internal nodes and generate constraints

Solve by substitution

Graph for f(g) = g(2)



Using a Polymorphic Function

- Function
  - $-$ fun f(g) = g(2);
  - $>$ val it = fn : (int $\rightarrow$ t) $\rightarrow$ 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

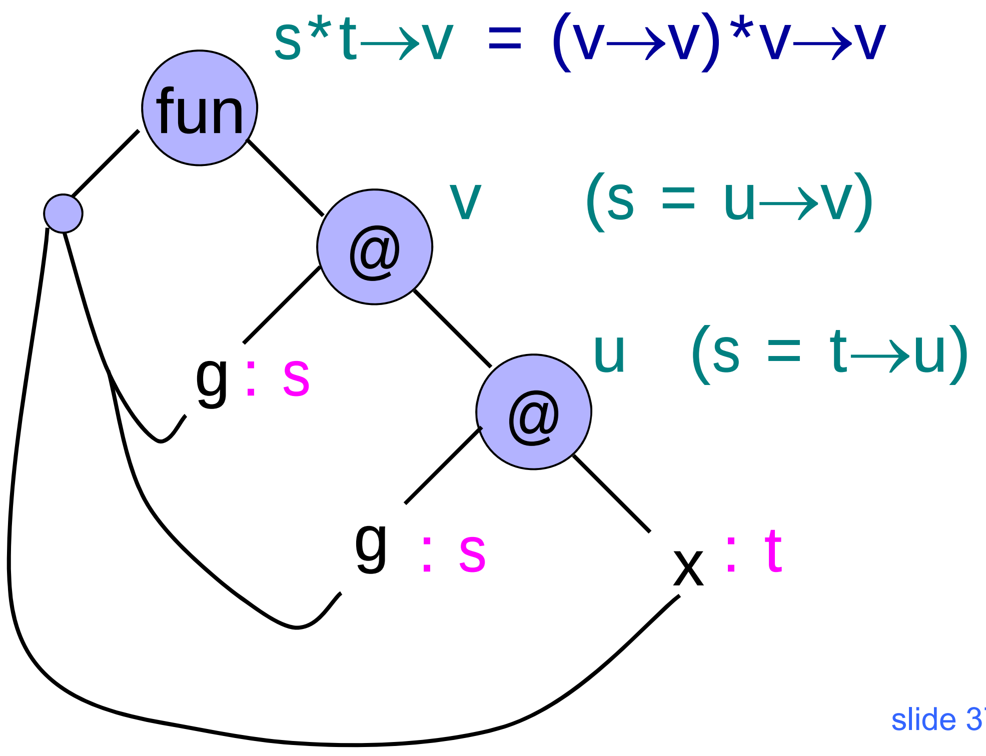
- Function definition
  - fun f(g,x) = g(g(x));
  - > val it = fn : (t → t)*t → t
- Type inference

Assign types to leaves

Propagate to internal nodes and generate constraints

Solve by substitution

Graph for f(g,x) = g(g(x))



Polymorphic Datatypes

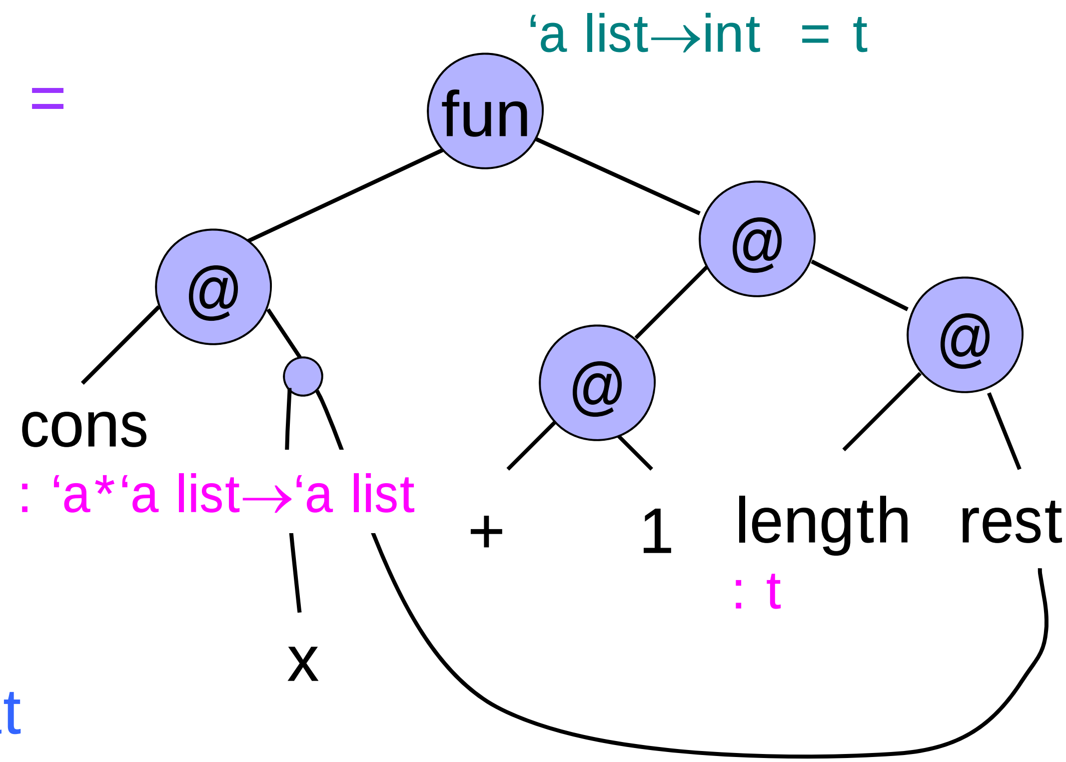
- Datatype with type variable ’a is syntax for “type variable a”
  - datatype ‘a list = nil | cons of ‘a*(‘a list)
  - > nil : ‘a list
  - > cons : ‘a*(‘a list) → ‘a list
- Polymorphic function
  - fun length nil = 0
  - | length (cons(x,rest)) = 1 + length(rest)
  - > length : ‘a list → int
- Type inference
  - 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 is equal to the type of function name



Tricky, isn't it?

Type Inference Summary

- Type of expression computed, not declared
  - Does not require type declarations for variables
  - Find <u>most general type</u> by solving constraints
  - Leads to polymorphism
- Static type checking without type specifications
  - Idea can be applied to other program properties
- Sometimes provides better error detection than type checking
  - Type may indicate a programming error even if there is no type error (how?)

Costs of Type Inference

- More difficult to identify program line that causes error
- ML requires different syntax for values of different types
  - integer: 3, real: 3.0
- Complications with assignment took years to work out

Information From Type Inference

- An interesting function on lists

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

- Most general type

  reverse : ‘a list $\rightarrow$ ‘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”

See Koenig paper on course website

Param. Polymorphism: ML vs. C++

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

ML also has module system with explicit type parameters

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 <u>one function</u>
  - Typechecker determines how function can be used
- C++
  - Swap is compiled into linkable format
  - Linker <u>duplicates code</u> for each type of use
- Why the difference?
  - ML reference cell is passed by pointer, local x is a 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 and Type Inference

- 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
  - Aside: general overloading is NP-complete

Summary

- Types are important in modern languages
  - Organize and document the program, prevent 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 when program is compiled