CS 242

Types

John Mitchell

Type

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

- Examples
  - Integers
  - Strings
  - int → bool
  - (int → int) →bool
- "Non-examples"
  - {3, true, λx.x}
  - 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
    - 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

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
  - just as much an error as x() above

General definition of type error

- A *type error* occurs when execution of program is not faithful to the intended semantics
- Do you like this definition?
  - 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

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
  - Compile-time checking restricts program flexibility
    - Lisp list: elements can have different types
    - ML list: all elements must have same type

Expressiveness

- In Lisp, we can write function like
  - (lambda (x) (cond ((less x 10) x) (T (car x))))
  - Some uses will produce type error, some will not
- Static typing always conservative
  - if (big-hairy-boolean-expression)
    - then ((lambda (x) …) 5)
    - else ((lambda (x) …) 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, and Java
  - Lisp, Smalltalk: 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.

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

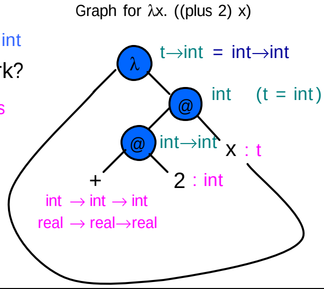
- Example
  - \- fun f(x) = 2+x;
  - > val it = fn : int → int
- How does this work?

Assign types to leaves

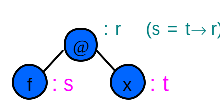
Propagate to internal nodes and generate constraints

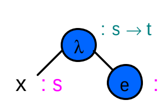
Solve by substitution

Graph for λx. ((plus 2) x)



Application and Abstraction



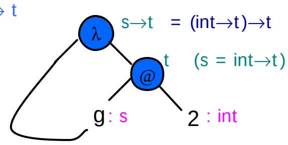


- Application
  - f must have function type domain→ range
  - domain of f must be type of argument x
  - result type is range of f
- Function expression
  - Type is function type domain→ range
  - Domain is type of variable x
  - Range is type of function body e

Types with type variables

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

Graph for λg. (g 2)



Assign types to leaves

Propagate to internal nodes and generate constraints

Solve by substitution

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

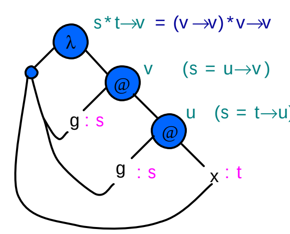
- Function
  - fun f(g) = g(2);
  - > val it = fn : (int → t) → t
- Incorrect use
  - fun not(x) = if x then false else true;
  - > val it = fn : bool → bool
  - f(not);

  Type error: cannot make bool → bool = int → t

Another Type Inference Example

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

Graph for λ⟨g,x⟩. g(g x)



Assign types to leaves

Propagate to internal nodes and generate constraints

Solve by substitution

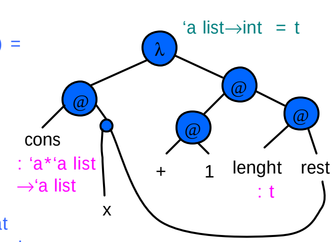
Polymorphic Datatypes

- 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
- 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 of "reverse"

Compare C++ templates

- Sec 6.4.1 – Parametric polymorphism
- Sec 6.4.2 – Implementation of parametric poly

Polymorphism vs Overloading

- Parametric polymorphism
  - Single algorithm may be given many types
  - Type variable may be replaced by *any* type
  - f : t→t => f : int→int, f : bool→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→int, real*real→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

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& , T& y){
    T tmp = x;  x=y;  y=tmp;
}
```

Declarations look similar, but compiled is 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;
  - > Error: overloaded variable cannot be resolved: +
- Why is a unique type needed?
  - Need to compile code ⇒ need to know which +
  - Efficiency of type inference
  - Aside: General overloading is NP-complete

    Two types, *true* and *false*

    Overloaded functions

    and : {*true***true*→*true*, *false***true*→*false*, ...}

Main Points about ML

- General-purpose procedural language
  - We have looked at "core language" only
  - Also: abstract data types, modules, concurrency,....
- Well-designed type system
  - Type inference
  - Polymorphism
  - Reliable -- no loopholes
  - Limited overloading