Types

John Mitchell

Reading: Chapter 6

Announcements

• Homework 1 due today
  – Turn in at the end of class, OR
  – Turn in by 5PM to the homework drop box
    • First floor of Gates Bldg outside Gates 182
    • Box is labeled CS 242 Homework

• Homework late policy
  – You may turn in up to 3 late homeworks this quarter
  – Each must be turned in by 5PM on Thursday
    • Same drop box as Wednesday homework

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 → bool
  • (int → int) → bool

◆“Non-examples”
  • {3, true, x, x}
  • Even integers
  • f: int → int if x>3 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
  
  ```javascript
  f(x) { make sure f is a function before calling f(x) }
  ```

- ML uses compile-time type checking
  
  ```ml
  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
  
  JavaScript array: elements can have different types
  
  ML: all elements must have same type
  
  - Which gives better programmer diagnostics?

### Expressiveness

- In JavaScript, we can write function like
  
  ```javascript
  function f(x) { return x < 10 ? x : x; }
  ```
  
  Some uses will produce type error, some will not

- Static typing always conservative
  
  ```javascript
  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.
  - 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
  
  ```ml
  int f(x) { return x+1; }
  int g(int y) { return f(y+1)*2; }
  ```

  - Look at body of each function and use declared types of identifiers to check agreement.

- Type inference
  
  ```ml
  int f(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
  
  ```ml
  - 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 \to int
- How does this work?
  - Assign types to leaves
  - Propagate to internal nodes and generate constraints
  - Solve by substitution

Graph for \( \lambda x. ((\text{plus} \ 2) \ x) \)

Application and Abstraction

- Application
  - \( f \) must have function type \( \text{domain} \to \text{range} \)
  - \( \text{domain} \) of \( f \) must be type of argument \( x \)
  - result type is range of \( f \)
- Function expression
  - Type is function type \( \text{domain} \to \text{range} \)
  - 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 \to t) \to t
- How does this work?
  - Assign types to leaves
  - Propagate to internal nodes and generate constraints
  - Solve by substitution

Graph for \( \lambda g. (g \ 2) \)

Use of Polymorphic Function

- Function
  - fun \( f(g) = g(2); \)
  - val it = fn : (int \to t) \to t
- Possible applications
  - fun \( \text{isEven}(x) = \ldots; \)
  - val it = fn : int \to bool
  - fun \( \text{add}(x) = 2+x; \)
  - val it = fn : int \to int
  - fun \( \text{f}(x); \)
  - val it = 4 : int

Another Type Inference Example

- Function Definition
  - fun \( f(g,x) = g(g(x)); \)
  - val it = fn : (t \to t)^*t \to t
- Type Inference
  - Assign types to leaves
  - Propagate to internal nodes and generate constraints
  - Solve by substitution
Polymorphic Datatypes

- 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 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)

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
  - Idea can be applied to other program properties
- Sometimes better error detection than type checking
  - Type may indicate a programming error even if there is no type error
    (example following slides)
- Some costs
  - More difficult to identify program line that causes error
  - ML requires different syntax for integer 3, real 3.0
  - Complications regarding 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 → '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 “Reading” page of CS242 site

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, 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

We do not expect you to master this.
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++)
    }

- **ML**
  - `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 ⇒ need to know which +
- Efficiency of type inference

Aside: General overloading is NP-complete

Two types, true and false

Overloaded functions

- `true*true -> true, false*true -> false, ...`

ML Overloading

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