TYPES

Kathleen Fisher

Reading: “Concepts in Programming Languages”, Chapter 6

Thanks to John Mitchell for some of these slides.
We are looking for homework graders.

- If you are interested, send mail to cs242@cs.stanford.edu
- Need to be available approximately 5-9pm on Thursdays.

You’ll be paid Stanford’s hourly rate.

- We’ll provide food “of your choice.”
- Previous graders have really enjoyed it.
- Great way to really learn the material.
Outline

- General discussion of types
  - What is a type?
  - Compile-time vs run-time checking
  - Conservative program analysis

- Type inference
  - Will study algorithm and examples
  - Good example of static analysis algorithm

- Polymorphism
  - Uniform vs non-uniform impl of polymorphism
  - Polymorphism vs overloading
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?
A type is a collection of computable values that share some structural property.

- **Examples**
  - Integer
  - String
  - Int → Bool
  - (Int → Int) → Bool

- **Non-examples**
  - \{3, True, \(\lambda x \rightarrow 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
JavaScript and Lisp use run-time type checking

\[ f(x) \]

Make sure \( f \) is a function before calling \( f \).

ML and Haskell use compile-time type checking

\[ f(x) \]

Must have \( f : A \rightarrow B \) and \( x : A \)

Basic tradeoff

- Both kinds of checking prevent type errors.
- Run-time checking slows down execution.
- Compile-time checking restricts program flexibility.
  - JavaScript array: elements can have different types
  - Haskell list: all elements must have same type
- Which gives better programmer diagnostics?
Expressiveness

- In JavaScript, we can write a 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(15);
  ```

  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, Smalltalk, ML, Haskell, Java, JavaScript
  - Dynamically typed: Lisp, Smalltalk, JavaScript
  - Statically typed: ML, Haskell, Java
Type Checking vs. Type Inference

- **Standard type checking:**
  ```c
  int f(int x) { return x+1; }
  int g(int y) { return f(y+1)*2; }
  ```
  - Examine body of each function.
  - Use declared types to **check** agreement.

- **Type inference:**
  ```c
  int f(int x) { return x+1; }
  int g(int y) { return f(y+1)*2; }
  ```
  - Examine code **without** type information. **Infer** the most general types that could have been declared.

*ML and Haskell are designated to make type inference feasible.*
Why study type inference?

- Types and type checking
  - Improved steadily since Algol 60
    - Eliminated sources of unsoundness.
    - Become substantially more expressive.
  - Important for modularity, reliability and compilation

- Type inference
  - Reduces syntactic overhead of expressive types
  - Guaranteed to produce *most general type*.
  - Widely regarded as important language innovation
  - Illustrative example of a flow-insensitive static analysis algorithm
History

- Original type inference algorithm was invented by Haskell Curry and Robert Feys for the simply typed lambda calculus in 1958.
- In 1969, Hindley extended the algorithm to a richer language and proved it always produced the most general type.
- In 1978, Milner independently developed equivalent algorithm, called algorithm W, during his work designing ML.
- In 1982, Damas proved the algorithm was complete.
- Already used in many languages: ML, Ada, Haskell, C# 3.0, F#, Visual Basic .Net 9.0, and soon in: Fortress, Perl 6, C++0x
- We’ll use ML to explain the algorithm because it is the original language to use the feature and is the simplest place to start.
ML Type Inference

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

- What is the type of `f`?
  - `+` has two types: `int → int → int`, `real → real → real`
  - `2` has only one type: `int`
  - This implies `+ : int → int → int`
  - From context, we need `x:int`
  - Therefore `f(x) = 2+x` has type `int → int`
Example

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

>val it = fn:int → int

What is the type of $f$?

Assign types to leaves

Propagate to internal nodes and generate constraints

Solve by substitution

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

$
\begin{array}{c}
\lambda \\
\int \to \int \\
\int \to \int
\end{array}

\begin{array}{c}
\int \\
(t = \int)
\end{array}

\begin{array}{c}
\int \to \int \\
\int \to \int
\end{array}

\begin{array}{c}
\text{2:} \int \\
\text{real} \to \text{real} \to \text{real}
\end{array}

\begin{array}{c}
\text{int} \\
\text{real}
\end{array}$
Apply function \( f \) to argument \( x \): \( f(x) \)

- Because \( f \) is being applied, its type (\( s \) in figure) must be a function type: domain \( \rightarrow \) range.
- Domain of \( f \) must be type of argument \( x \) (\( d \) in figure).
- Range of \( f \) must be result type of expression (\( r \) in figure).
- Solving, we get: \( s = d \rightarrow r \).
Abstraction

- Function expression: \( \lambda x \rightarrow e \)
  - Type of lambda abstraction (\( s \) in figure) must be a function type: domain \( \rightarrow \) range.
  - Domain is type of abstracted variable \( x \) (\( d \) in figure).
  - Range is type of function body \( e \) (\( r \) in figure).
  - Solving, we get: \( s = d \rightarrow r \).
Types with Type Variables

- **Example**
  ```
  -fun f(g) = g(2);
  >val it = fn : (int → t) → t
  ```

- **What is the type of** \( f \)?

  Assign types to leaves

  Propagate to internal nodes and generate constraints

  Solve by substitution

  Graph for \( \lambda g \rightarrow (g \ 2) \)
  ```
  \( \lambda \)  
  \( s \rightarrow t = (\text{int} \rightarrow t) \rightarrow t \)  
  \( t \)  
  \( (s = \text{int} \rightarrow t) \)  
  \( g : s \)  
  \( 2 : \text{int} \)
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);`
  
  **Error:** operator and operand don't agree
  - operator domain: `int` → `'Z`
  - operand: `bool` → `bool`
  
  **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**
  - Solve by substitution
  - 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**
  - `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

\[
\text{length} \left( \text{cons}(x, \text{rest}) \right) = 1 + \text{length} \left( \text{rest} \right)
\]
Type Inference with Recursion

- \( \text{length}(\text{cons}(x, \text{rest})) = 1 + \text{length}(\text{rest}) \)

Collected Constraints:

\[
\begin{align*}
p &= \top \\
p &= v \rightarrow w \\
\text{int} \rightarrow \text{int} &= r \rightarrow w \\
t &= u \rightarrow r \\
'a^\ast 'a \text{ list} \rightarrow 'a \text{ list} &= s \ast u \rightarrow v
\end{align*}
\]
Type Inference with Recursion

- \( \text{length}(\text{cons}(x, \text{rest})) = 1 + \text{length}(\text{rest}) \)

Collected Constraints:
- \( p = t \)
- \( p = v \rightarrow w \)
- \( \text{int} \rightarrow \text{int} = r \rightarrow w \)
- \( t = u \rightarrow r \)
- \( 'a = s \)
- \( 'a \text{ list} = u \)
- \( 'a \text{ list} = v \)
Type Inference with Recursion

- \( \text{length(cons(x, rest))} = 1 + \text{length(rest)} \)

Collected Constraints:
- \( p = \top \)
- \( p = \text{'a list \rightarrow w} \)
- \( \text{int \rightarrow int = r \rightarrow w} \)
- \( t = \text{'a list \rightarrow r} \)
Type Inference with Recursion

- \( \text{length}(\text{cons}(x, \text{rest})) = 1 + \text{length}(\text{rest}) \)

Collected Constraints:
- \( p = \top \)
- \( p = \text{'a list} \rightarrow \text{int} \)
- \( t = \text{'a list} \rightarrow \text{int} \)

Result:
- \( p = \text{'a list} \rightarrow \text{int} \)
Multiple Clauses

- Function with multiple clauses
  - fun append(nil,l) = l
  - append(x::xs,l) = x :: append(xs,l)
  > append: 'a list * 'a list → int

- Infer type of each branch
  - First branch:
    append : 'a list * 'b → 'b
  - First branch:
    append : 'a list * 'b → 'a list

- Combine by equating types of two branches:
  append : 'a list * 'a list → 'a list
Most General Type

- Type inference is **guaranteed** to produce the **most general type**: 
  
  ```haskell
  fun map(f,nil) = nil
  | map(f, x::xs) = f(x) :: (map(f,xs))
  > map:('a → 'b) * 'a list → 'b list
  ```

- Function has many other, less general types:
  ```haskell
  - map:('a → int) * 'a list → int list
  - map:(bool → 'b) * bool list → 'b list
  - map:(char → int) * char list → int list
  ```

- Less general types are all **instances** of most general type, also called the **principal type**.
When the Hindley/Milner type inference algorithm was developed, its complexity was unknown.

In 1989, Mairson proved that the problem was exponential-time complete.

Tractable in practice though…
Consider this function...

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

... and its most general type:

```
reverse : 'a list → 'b list
```

What does this type mean?

Reversing a list does not change its type, so there must be an error in the definition of `reverse`!

See Koenig paper on “Reading” page of CS242 site
Type Inference: Key Points

- Type inference computes the types of expressions
  - Does not require type declarations for variables
  - Finds the *most general type* by solving constraints
  - Leads to polymorphism

- Sometimes better error detection than type checking
  - Type may indicate a programming error even if no type error.

- Some costs
  - More difficult to identify program line that causes error
  - ML requires different syntax for integer 3, real 3.0.
  - Natural implementation requires uniform representation sizes.
  - Complications regarding assignment took years to work out.

- Idea can be applied to other program properties
  - Discover properties of program using same kind of analysis
Haskell Type Inference

- Haskell also uses Hindley Milner type inference.
- Haskell uses type classes to support user-defined overloading, so the inference algorithm is more complicated.
- ML restricts the language to ensure that no annotations are required, ever.
- Haskell provides various features like polymorphic recursion for which types cannot be inferred and so the user must provide annotations.
Parametric Polymorphism: ML vs C++

- **ML polymorphic function**
  - Declarations require no type information.
  - Type inference uses type variables to type expressions.
  - Type inference substitutes for variables as needed to instantiate polymorphic code.

- **C++ function template**
  - Programmer must declare the argument and result types of functions.
  - Programmers must use explicit type parameters to express polymorphism.
  - 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 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. The local \( x \) is a pointer to value on heap, so its size is constant.
  - C++ arguments passed by reference (pointer), but local \( x \) is on the stack, so its size depends on the type.
Another Example

- C++ polymorphic sort function
  ```cpp
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++)
}
```

- What parts of code depend on the type?
  - Indexing into array
  - Meaning and implementation of `<`
Polymorphism vs Overloading

- **Parametric polymorphism**
  - Single algorithm may be given *many* types
  - Type variable may be replaced by *any* type
  - if $f : t \rightarrow t$ then $f : \text{int} \rightarrow \text{int}$, $f : \text{bool} \rightarrow \text{bool}$, ...

- **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 $\text{int} \times \text{int} \rightarrow \text{int}$, $\text{real} \times \text{real} \rightarrow \text{real}$, *no others*
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, so need to know which +
- Efficiency of type inference
- Aside: General overloading is NP-complete
  Two types, true and false
  Overloaded functions
  and : {true*x true→true, false*x true→false, ...}
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
  - One symbol with multiple meanings, resolved at compile time