TYPES

Kathleen Fisher

Reading: "Concepts in Programming Languages", Chapter 6
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?
What is a type?

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, \(x\rightarrow x\)}
  - Even integers
  - \(\{f:\text{Int} \rightarrow \text{Int} \mid x>3 \Rightarrow f(x) > x \cdot (x+1)\}\)

Distinction between sets of values 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
What is a type error?

- A **type error** is something the compiler/interpreter reports when I make a mistake in my syntactically correct program?

- Languages represent values as sequences of bits. A **type error** occurs when a bit sequence written for one type is used as a bit sequence for another type?

- A **type error** occurs when a value is used in a way inconsistent with its definition.
Type errors are language dependent

- **Array out of bounds access**
  - C/C++: runtime errors.
  - Haskell/Java: dynamic type errors.

- **Null pointer dereference**
  - C/C++: null pointer dereferences are run-time errors.
  - In Haskell/ML, pointers are hidden inside datatypes. Null pointer dereferences correspond to incorrect use of these datatypes. Such errors are static type errors.
Compile-time vs Run-time Checking

- JavaScript and Lisp use run-time type checking
  \[ f(x) \] Make sure \( f \) is a function \textit{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
  // f(int x) { return x+1; }
  // 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 designed** 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
Subset of Haskell to explain type inference.

Haskell and ML both have overloading, which slightly complicates type inference. We won’t worry about type inference with overloading.

```
<decl> ::= [<name> <pat> = <exp>]
<pat> ::= Id | (<pat>, <pat>) | <pat> : <pat> | []
<exp> ::= Int | Bool | [] | Id | (<exp>)
       | <exp> <op> <exp>
       | <exp> <exp> | (<exp>, <exp>)
       | if <exp> then <exp> else <exp>
```
Type Inference: Basic Idea

- Example

\[ f \ x = 2 + x \]
\[ \text{> } f :: \text{Int} \rightarrow \text{Int} \]

- What is the type of \( f \)?
  - \( + \) has type: \( \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \)
  - \( 2 \) has type: \( \text{Int} \)
  - Since we are applying \( + \) to \( x \) we need \( x :: \text{Int} \)
  - Therefore \( f \ x = 2 + x \) has type \( \text{Int} \rightarrow \text{Int} \)

Overloaded functions introduce more cases to consider.
Step 1: Parse Program

- Parse program text to construct parse tree.

$f \ x = 2 + x$

Infix operators are converted to normal function application during parsing:

$2 + x \rightarrow (+) 2 \ x$
Step 2: Assign type variables to nodes

Variables are given same type as binding occurrence.
Step 3: Add Constraints

\[ f(x) = 2 + x \]

\[
\begin{align*}
\text{Fun} \\
\ t_0 &= \ t_1 \rightarrow \ t_6 \\
\ t_4 &= \ t_1 \rightarrow \ t_6 \\
\ t_2 &= \ t_3 \rightarrow \ t_4 \\
\ t_2 &= \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\ t_3 &= \text{Int} \\
\end{align*}
\]
Step 4: Solve Constraints

\[
\begin{align*}
t_0 &= t_1 \to t_6 \\
t_4 &= t_1 \to t_6 \\
t_2 &= t_3 \to t_4 \\
t_2 &= \text{Int} \to \text{Int} \to \text{Int} \\
t_3 &= \text{Int} \\
\end{align*}
\]
Step 5: Determine type of declaration

\[
\begin{align*}
  t_0 &= \text{Int} \rightarrow \text{Int} \\
  t_1 &= \text{Int} \\
  t_6 &= \text{Int} \rightarrow \text{Int} \\
  t_4 &= \text{Int} \rightarrow \text{Int} \\
  t_2 &= \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
  t_3 &= \text{Int}
\end{align*}
\]
Type Inference Algorithm

- Parse program to build parse tree
- Assign type variables to nodes in tree
- Generate constraints:
  - From environment: constants (2), built-in operators (+), known functions (\texttt{tail}).
  - From form of parse tree: e.g., application and abstraction nodes.
- Solve constraints using \texttt{unification}.
- Determine types of top-level declarations.
Apply function $f$ to argument $x$:

- Because $f$ is being applied, its type ($t_0$ in figure) must be a function type: $\text{domain} \rightarrow \text{range}$.
- Domain of $f$ must be type of argument $x$ ($t_1$ in figure).
- Range of $f$ must be result type of expression ($t_2$ in figure).
- Hence we get the constraint: $t_0 = t_1 \rightarrow t_2$
Function declaration:
- Type of function $f$ ($t_0$ in figure) must be a function type: domain $\rightarrow$ range.
- Domain is type of abstracted variable $x$ ($t_1$ in figure).
- Range is type of function body $e$ ($t_2$ in figure).
- Hence we get the constraint: $t_0 = t_1 \rightarrow t_2$. 
Inferring Polymorphic Types

- Example:
  \[
  f \cdot g = g \cdot 2 \\
  \quad > f :: (\text{Int} \rightarrow t_4) \rightarrow t_4
  \]

- Step 1:
  Build Parse Tree
Inferring Polymorphic Types

- Example:
  \[ f \circ g = g \circ 2 \]

- Step 2:
  Assign type variables
Inferring Polymorphic Types

- **Example:**
  ```plaintext
  f \ g = g \ 2
  > f :: (Int \rightarrow \ t_4) \rightarrow \ t_4
  ```

- **Step 3:**
  Generate constraints

  ```plaintext
  t_0 = t_1 \rightarrow \ t_4
  t_1 = t_3 \rightarrow \ t_4
  t_3 = \text{Int}
  ```
Inferring Polymorphic Types

- Example:
  \[ f \circ g = g \circ 2 \]
  \[ > f :: (\text{Int} \to t_4) \to t_4 \]

- Step 4:
  Solve constraints

\[
\begin{align*}
t_0 &= t_1 \to t_4 \\
t_1 &= t_3 \to t_4 \\
t_3 &= \text{Int}
\end{align*}
\]
Inferring Polymorphic Types

- Example:
  \[
  f \ g = g \ 2 \\
  > f :: (\text{Int} \rightarrow t_4) \rightarrow t_4
  \]

- Step 5:
  Determine type of top-level declaration

Unconstrained type variables become polymorphic types.

\[
\begin{align*}
  t_0 &= (\text{Int} \rightarrow t_4) \rightarrow t_4 \\
  t_1 &= \text{Int} \rightarrow t_4 \\
  t_3 &= \text{Int}
\end{align*}
\]
Using Polymorphic Functions

- **Function:**
  
  \[
  f \circ g = g \circ 2 \\
  f :: (\text{Int} \rightarrow t_4) \rightarrow t_4
  \]

- **Possible applications:**

  - \[
    \text{add} \ x = 2 + x \\
    > \text{add} :: \text{Int} \rightarrow \text{Int}
    \]

  - \[
    f \ \text{add} \\
    > 4 :: \text{Int}
    \]

  - \[
    \text{isEven} \ x = \text{mod} \ (x, 2) == 0 \\
    > \text{isEven} :: \text{Int} \rightarrow \text{Bool}
    \]

  - \[
    f \ \text{isEven} \\
    > \text{True} :: \text{Int}
    \]
Recognizing Type Errors

- **Function**
  
  \[
  f \circ g = g \ 2 \\
  > f :: (\text{Int} \rightarrow t) \rightarrow t
  \]

- **Incorrect use**

  \[
  \text{not } x = \text{if } x \text{ then True else False} \\
  > \text{not :: Bool} \rightarrow \text{Bool}
  \]

  \[
  f \ \text{not} \\
  > \text{Error: operator and operand don’t agree} \\
  \text{operator domain: Int} \rightarrow a \\text{operator domain: Bool} \rightarrow \text{Bool}
  \]

- **Type error:**

  cannot unify \text{Bool} \rightarrow \text{Bool} and \text{Int} \rightarrow t
Another Example

- Example:
  \[ f \ (g, x) = g \ (g \ x) \]
  \[ > f :: (t_8 \rightarrow t_8, t_8) \rightarrow t_8 \]

- Step 1:
  Build Parse Tree
Another Example

- Example:
  \[ f(g,x) = g(g(x)) \]
  \[ > f :: (\text{t}_8 \rightarrow \text{t}_8, \text{t}_8) \rightarrow \text{t}_8 \]

- Step 2: Assign type variables
Another Example

- Example:
  \[ f \left( g, x \right) = g \left( g \left( x \right) \right) \]
  \[ > f :: (t_8 \to t_8, t_8) \to t_8 \]

- Step 3: Generate constraints

\[
\begin{align*}
t_0 &= t_3 \to t_8 \\
t_3 &= (t_1, t_2) \\
t_1 &= t_7 \to t_8 \\
t_1 &= t_2 \to t_7
\end{align*}
\]
Another Example

- Example:
  \[ f (g, x) = g (g x) \]
  \[ f :: (t_8 \rightarrow t_8, t_8) \rightarrow t_8 \]

- Step 4: Solve constraints

\[ t_0 = t_3 \rightarrow t_8 \]
\[ t_3 = (t_1, t_2) \]
\[ t_1 = t_7 \rightarrow t_8 \]
\[ t_1 = t_2 \rightarrow t_7 \]

\[ t_0 = (t_8 \rightarrow t_8, t_8) \rightarrow t_8 \]
Another Example

- Example:
  
  \[ f \ (g, x) = g \ (g \ x) \]
  
  > \( f \ :: \ (t_8 \rightarrow t_8, \ t_8) \rightarrow t_8 \)

- Step 5: Determine type of \( f \)

\[
\begin{align*}
t_0 &= t_3 \rightarrow t_8 \\
t_3 &= (t_1, t_2) \\
t_1 &= t_7 \rightarrow t_8 \\
t_1 &= t_2 \rightarrow t_7 \\
t_0 &= (t_8 \rightarrow t_8, \ t_8) \rightarrow t_8
\end{align*}
\]
Polymorphic Datatypes

- Often, functions over datatypes are written with multiple clauses:

  length [] = 0
  length (x:rest) = 1 + (length rest)

- Type inference
  - Infer separate type for each clause
  - Combine by adding constraint that the types of the branches must be equal.
Type Inference with Datatypes

- Example: \( \text{length (x:rest) = 1 + (length rest)} \)
- Step 1: Build Parse Tree
Type Inference with Datatypes

- Example: \[ \text{length } (x: \text{rest}) = 1 + (\text{length rest}) \]
- Step 2: Assign type variables
Example:

\[
\text{length } (x: \text{rest}) = 1 + (\text{length rest})
\]

Step 3: Gen. constraints

- \( t_0 = t_3 \rightarrow t_{10} \)
- \( t_3 = t_2 \)
- \( t_3 = [t_1] \)
- \( t_6 = t_9 \rightarrow t_{10} \)
- \( t_4 = t_5 \rightarrow t_6 \)
- \( t_4 = \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \)
- \( t_5 = \text{Int} \)
- \( t_0 = t_2 \rightarrow t_9 \)
Type Inference with Datatypes

- Example:
  \[
  \text{length } (x: \text{rest}) = 1 + (\text{length rest})
  \]

- Step 3: Solve Constraints

  \[
  t_0 = t_3 \rightarrow t_{10}
  
  t_3 = t_2
  
  t_3 = [t_1]
  
  t_6 = t_9 \rightarrow t_{10}
  
  t_4 = t_5 \rightarrow t_6
  
  t_4 = \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
  
  t_5 = \text{Int}
  
  t_0 = t_2 \rightarrow t_9
  \]

  \[
  t_0 = [t_1] \rightarrow \text{Int}
  \]
Multiple Clauses

- Function with multiple clauses
  
  \[
  \text{append} \hspace{1em} (\text{[]} , r) = r \\
  \text{append} \hspace{1em} (x : xs , r) = x : \text{append} \hspace{1em} (xs , r)
  \]

- Infer type of each branch
  - First branch:
    - \( \text{append} :: ([t_1] , t_2) \rightarrow t_2 \)
  
  - Second branch:
    - \( \text{append} :: ([t_3] , t_4) \rightarrow [t_3] \)

- Combine by equating types of two branches:
  - \( \text{append} :: ([t_1] , [t_1]) \rightarrow [t_1] \)
Most General Type

- Type inference is **guaranteed** to produce the **most general type**:

  ```
  \text{map} (f, \mathbb{\text{[]}}) = \mathbb{\text{[]}}
  \text{map} (f, x:xs) = f \times : \text{map} (f, xs)
  > \text{map} :: (t_1 -> t_2, \mathbb{[t_1]}) -> \mathbb{[t_2]}
  ```

- Function has many other, less general types:

  ```
  > \text{map} :: (t_1 -> \text{Int}, \mathbb{[t_1]}) -> \mathbb{[\text{Int}]}
  > \text{map} :: (\text{Bool} -> t_2, \mathbb{[\text{Bool}]}) -> \mathbb{[t_2]}
  > \text{map} :: (\text{Char} -> \text{Int}, \mathbb{[\text{Char}]}) -> \mathbb{[\text{Int}]}
  ```

- 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…
Information from Type Inference

- Consider this function...
  
  \[
  \begin{align*}
    \text{reverse } [] &= [] \\
    \text{reverse } (x:xs) &= \text{reverse } xs
  \end{align*}
  \]

- ... and its most general type:

  \[> \text{reverse} :: [t_1] \rightarrow [t_2]\]

- What does this type mean?

  Reversing a list does not change its type, so there must be an error in the definition of \text{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.
  - 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 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: Haskell vs C++

- **Haskell polymorphic function**
  - Declarations (generally) 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.
Example: Swap Two Values

- Haskell

```
swap :: (IORef a, IORef a) -> IO ()
swap (x,y) = do {
  val_x <- readIORef x; val_y <- readIORef y;
  writeIORef y val_x;  writeIORef x val_y;
  return ()}
```

- C++

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

Declarations both swap two values polymorphically, but they are compiled very differently.
Implementation

- **Haskell**
  - *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?**
  - Haskell 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
  - In ML, + has types $\text{int} * \text{int} \rightarrow \text{int}$, $\text{real} * \text{real} \rightarrow \text{real}$, no others
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