301AA - Advanced Programming [AP-2017]

Lecturer: Andrea Corradini andrea@di.unipi.it
Tutor: Lillo Galletta galletta@di.unipi.it

Department of Computer Science, Pisa
Academic Year 2017/18

AP-2017-19: Type inference
Type Inference

• **Type inference**: the process of associating a type with each symbol of a program, if possible, ensuring type safety.

• Recalling several relevant definitions:
  – (Data) types
  – Type system
  – Type error
  – Type safety
  – Dynamic vs. static type checking
Data types

• A **(data) type** is a *homogeneous collection of values, effectively presented, equipped with a set of operations* which manipulate these values.

• Various perspectives:
  – collection of values from a “domain” (the *denotational* approach)
  – internal structure of a bunch of data, described down to the level of a small set of fundamental types (the *structural* approach)
  – collection of well-defined operations that can be applied to objects of that type (the *abstraction* approach)
A type system consists of
1. The set of predefined types of the language.
2. The mechanisms which permit the definition of new types.
3. The mechanisms for the control (checking) of types, which include:
   1. Equivalence rules which specify when two formally different types correspond to the same type.
   2. Compatibility rules specifying when a value of a one type can be used in given context.
   3. Rules and techniques for type inference which specify how the language assigns a type to a complex expression based on information about its components (and sometimes on the context).
4. The specification as to whether (or which) constraints are statically or dynamically checked.
Type errors

• A **type error** occurs when a value is used in a way that is inconsistent with its definition

• Implementations can react in various ways
  – Hardware interrupt, *e.g.* apply fp addition to non-legal bit configuration
  – OS exception, *e.g.* segmentation fault when dereferencing 0 in C
  – Continue execution possibly with wrong values

• Examples
  – Array out of bounds access
    • C/C++: runtime errors
    • Java: dynamic type error
  – Null pointer dereferencing
    • C/C++: run-time errors
    • Java: dynamic type error
    • Haskell/ML: pointers are hidden inside datatypes
      – Null pointer dereferences would be incorrect use of these datatypes, therefore static type errors
Type safety

• A language is **type safe (strongly typed)** when no program can violate the distinctions between types defined in its type system
  
• That is, when no program, during its execution, can generate an **unsignalled type error**

• Also: if code accesses data, it is handled with the type associated with the creation and previous manipulation of that data
Type checking

• To prevent type errors, before any operation is performed, its operands must be type-checked to ensure that they comply with the compatibility rules of the type system
  – E.g., indexing operation: check that the left operand is an array, and that the right operand is a value of the array’s index type.

• **Statically typed** languages: (most) type checking is done during compilation

• **Dynamically typed** languages: type checking is done at runtime
Static vs dynamic typing

- In a **statically typed** PL:
  - all variables and expressions have fixed types (either stated by the programmer or inferred by the compiler)
  - most operands are type-checked at *compile-time*.

- Most PLs are called “statically typed”, including Ada, C, C++, Java, Haskell, ... even if some type-checking is done at run-time (e.g. access to arrays)

- In a **dynamically typed** PL:
  - values have fixed types, but variables and expressions do not
  - operands must be type-checked when they are computed at *run-time*.

- Some PLs and many scripting languages are dynamically typed, including Smalltalk, Lisp, Prolog, Perl, Python.
Polymorphism in Haskell

- Coercion
  - Implicit
  - Explicit
- Parametric
  - Bounded
    - Covariant
    - Invariant
    - Contravariant
- Inclusion
- Overriding
  - Type Inference
    - Type classes
- Overloading
- Universal
- Ad hoc

Polymorphism
Universal vs Ad-hoc Polymorphism

• Universal polymorphism
  – *Single algorithm* may be given many types
  – Type variables (implicit or explicit) may be replaced by any type (almost... -> *bounded polymorphism*)
  – `head::[a]→a` thus `head::[Int]→Int`, `head:[Bool]→Bool`, ...

• Ad-hoc polymorphism (overloading)
  – A single symbol may refer to more than one algorithm.
  – Each algorithm may have different type.
  – Choice of algorithm determined by type context.
  – `+` has types `Int → Int → Int` and `Float → Float → Float`, but not `t→t→t` for arbitrary `t`. 
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 *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.
  – Increasingly used also in imperative/OO languages.
  – Illustrative example of a flow-insensitive static analysis algorithm.
uHaskell

• Subset of Haskell to explain type inference.
  – Will do not consider overloading now

\[
<\text{decl}> ::= <\text{name}> <\text{pat}> = <\text{exp}>
\]
\[
<\text{pat}> ::= \text{Id} \mid (<\text{pat}>, <\text{pat}>) \mid <\text{pat}> : <\text{pat}> \mid []
\]
\[
<\text{exp}> ::= \text{Int} \mid \text{Bool} \mid [] \mid \text{Id} \mid (<\text{exp}>)
\]
\[
\mid <\text{exp}> \text{ op } <\text{exp}>
\]
\[
\mid <\text{exp}> <\text{exp}> \mid (<\text{exp}>, <\text{exp}>)
\]
\[
\mid \text{if } <\text{exp}> \text{ then } <\text{exp}> \text{ else } <\text{exp}>
\]
Type Inference: Basic Idea

• Example

\[ f \ x = 2 + x \quad -- \text{a simple declaration} \]

• What is the type of \( f \)?
  
  – \( + \) has type: \( \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \)
    
    (with overloading would be \( \text{Num} \ a \rightarrow a \rightarrow a \rightarrow a \))
  
  – \( 2 \) has type: \( \text{Int} \)
  
  – Since we are applying \( + \) to \( x \) we need \( x :: \text{Int} \)
  
  – Therefore \( f \ x = 2 + x \) implies that \( f \) has type \( \text{Int} \rightarrow \text{Int} \)

\[ f \ x = 2 + x \quad -- \text{a simple declaration} \]

> \( f :: \text{Int} \rightarrow \text{Int} \)
Step 1: Parse Program

- Parse program text to construct parse tree.

- Binary @-nodes to represent application
- Ternary Fun-node for function definitions
- Infix operators are converted to Curried function application during parsing: \( 2 + x \rightarrow (+) 2 x \)
Step 2: Assign type variables to nodes

Variables are given same type as binding occurrence.
Constraints from Application Nodes

- Function application (apply $f$ to $x$)
  - Type of $f$ ($t_0$ in figure) must be $\text{domain} \rightarrow \text{range}$.
  - Domain of $f$ must be type of argument $x$ ($t_1$ in fig)
  - Range of $f$ must be result of application ($t_2$ in fig)
  - Constraint: $t_0 = t_1 \rightarrow t_2$
Constraints from Abstractions

- Function declaration:
  - Type of $f$ (t_0 in figure) must be $\text{domain} \rightarrow \text{range}$
  - Domain is type of abstracted variable $x$ (t_1 in fig)
  - Range is type of function body $e$ (t_2 in fig)
  - Constraint: $t_0 = t_1 \rightarrow t_2$
  - Idem for lambdas: $f = \lambda x \rightarrow e$
Step 3: Add Constraints

\[
t_0 = t_1 \rightarrow t_6 \\
t_4 = t_1 \rightarrow t_6 \\
t_2 = t_3 \rightarrow t_4 \\
t_2 = \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
t_3 = \text{Int}
\]
Step 4: Solve Constraints

\[ t_0 = t_1 \rightarrow t_6 \]
\[ t_4 = t_1 \rightarrow t_6 \]
\[ t_2 = t_3 \rightarrow t_4 \]
\[ t_2 = \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ t_3 = \text{Int} \]

\[ t_0 = t_1 \rightarrow t_6 \]
\[ t_4 = t_1 \rightarrow t_6 \]
\[ t_4 = \text{Int} \rightarrow \text{Int} \]
\[ t_2 = \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ t_3 = \text{Int} \rightarrow \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \]

\[ t_3 = \text{Int} \]
\[ t_4 = \text{Int} \rightarrow \text{Int} \]

\[ t_1 = \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ t_3 = \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ t_1 = \text{Int} \]
\[ t_6 = \text{Int} \]

\[ t_0 = \text{Int} \rightarrow \text{Int} \]
\[ t_1 = \text{Int} \]
\[ t_6 = \text{Int} \]
\[ t_4 = \text{Int} \rightarrow \text{Int} \]
\[ t_2 = \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ t_3 = \text{Int} \]

\[ t_1 = \text{Int} \]
\[ t_6 = \text{Int} \]

\[ t_4 = \text{Int} \rightarrow \text{Int} \]
Step 5: Determine type of declaration

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

\[ f :: \text{Int} \rightarrow \text{Int} \]

\[ t_0 = \text{Int} \rightarrow \text{Int} \]
\[ t_1 = \text{Int} \]
\[ t_6 = \text{Int} \]
\[ t_4 = \text{Int} \rightarrow \text{Int} \]
\[ t_2 = \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]
\[ t_3 = \text{Int} \]
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 \((\text{tail})\).
  – From form of parse tree: e.g., application and abstraction nodes.
• Solve constraints using \textit{unification}
• Determine types of top-level declarations
Inferring Polymorphic Types

Example:

Step 1:
Build Parse Tree

\[
\begin{align*}
f &\, g = g 2 \\
& > f :: (\text{Int} \to t_4) \to t_4
\end{align*}
\]
Inferring Polymorphic Types

- Example:
  \[
  f \ g = g \ 2 \\
  \Rightarrow f :: (Int \rightarrow t_4) \rightarrow t_4
  \]

- Step 2:
  Assign type variables
Inferring Polymorphic Types

• Example:

\[
\begin{align*}
\text{f g} &= \text{g 2} \\
\Rightarrow \text{f :: (Int -> t_4)} &\rightarrow t_4
\end{align*}
\]

• Step 3:

Generate constraints

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

• Example:

\[
\begin{align*}
\text{t}_0 & = \text{t}_1 \rightarrow \text{t}_4 \\
\text{t}_1 & = \text{t}_3 \rightarrow \text{t}_4 \\
\text{t}_3 & = \text{Int}
\end{align*}
\]

\[
f \ g = g\ 2 \\
> f :: (\text{Int} \rightarrow \text{t}_4) \rightarrow \text{t}_4
\]

• Step 4:
Solve constraints

\[
\begin{align*}
\text{t}_0 & = (\text{Int} \rightarrow \text{t}_4) \rightarrow \text{t}_4 \\
\text{t}_1 & = \text{Int} \rightarrow \text{t}_4 \\
\text{t}_3 & = \text{Int}
\end{align*}
\]
Inferring Polymorphic Types

• Example:

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

• Step 5:
Determine type of top-level declaration

Unconstrained type variables become polymorphic types.

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

• Function:

\[
\begin{align*}
  f \ g &= g \ 2 \\
  > f :: (\text{Int} \to t_4) \to t_4
\end{align*}
\]

• Possible applications:

\[
\begin{align*}
  \text{add} \ x &= 2 + x \\
  > \text{add} :: \text{Int} \to \text{Int}
\end{align*}
\]

\[
\begin{align*}
  f \ \text{add} \\
  > 4 :: \text{Int}
\end{align*}
\]

\[
\begin{align*}
  \text{isEven} \ x &= \text{mod} (x, 2) == 0 \\
  > \text{isEven} :: \text{Int} \to \text{Bool}
\end{align*}
\]

\[
\begin{align*}
  f \ \text{isEven} \\
  > \text{True} :: \text{Int}
\end{align*}
\]
Recognizing Type Errors

• Function:

\[
\begin{align*}
 f & \equiv g \\
 & > f :: (\text{Int} \rightarrow t_4) \rightarrow t_4
\end{align*}
\]

• Incorrect use

\[
\begin{align*}
\text{not } x & = \text{if } x \text{ then True else False} \\
& > \text{not} :: \text{Bool} \rightarrow \text{Bool} \\
f & \text{not} \\
& > \text{Error: operator and operand don't agree} \\
& \quad \text{operator domain: } \text{Int} \rightarrow a \\
& \quad \text{operand: } \text{Bool} \rightarrow \text{Bool}
\end{align*}
\]

• Type error: cannot unify \( \text{Bool} \rightarrow \text{Bool} \) and \( \text{Int} \rightarrow t \)
Polymorphic Datatypes

• Functions may have multiple clauses, and may be recursive

\[
\text{length } [] = 0 \\
\text{length } (x:rest) = 1 + (\text{length } rest)
\]

• Type inference
  – Infer separate type for each clause
  – Add constraint that all clauses must have the same type
  – Recursive calls: function has same type as its definition
Type Inference with Datatypes

• Example: $\text{length } (x: \text{rest}) = 1 + (\text{length } \text{rest})$

• Step 1: Build Parse Tree
Type Inference with Datatypes

• Example:  
  
  $$\text{length (x:rest) = 1 + (length rest)}$$

• Step 2: Assign type variables
Type Inference with Datatypes

• Example:

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

• Step 3: Generate 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:

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

• Step 3: Solve Constraints

\[
\begin{align*}
t_0 &= t_3 \rightarrow t_{10} \\
t_3 &= t_2 \\
t_3 &= [t_1] \\
t_6 &= t_9 \rightarrow t_{10} \\
t_4 &= t_5 \rightarrow t_6 \\
t_4 &= \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
t_5 &= \text{Int} \\
t_0 &= t_2 \rightarrow t_9
\end{align*}
\]
Multiple Clauses

• Function with multiple clauses

\[
\text{append } ([], r) = r \\
\text{append } (x:xs, r) = x : \text{append } (xs, r)
\]

• Infer type of each clause
  – First clause:
    \[
    > \text{append} :: ([t_1], t_2) \rightarrow t_2
    \]
  – Second clause:
    \[
    > \text{append} :: ([t_3], t_4) \rightarrow [t_3]
    \]

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

• Type inference produces the *most general type*

```plaintext
map (f, []) = []
map (f, x:xs) = f x : map (f, xs)
> map :: (t_1 -> t_2, [t_1]) -> [t_2]
```

• Functions may have many less general types

```plaintext
> map :: (t_1 -> Int, [t_1]) -> [Int]
> map :: (Bool -> t_2, [Bool]) -> [t_2]
> map :: (Char -> Int, [Char]) -> [Int]
```

• Less general types are all instances of most general type, also called the **principal type**
History

• Original type inference algorithm
  – 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.
  – Currently used in many languages: ML, Ada, Haskell, C# 3.0, F#, Visual Basic .Net 9.0. Have been plans for Fortress, Perl 6, C+ +0x,... A bit also in Java....
Complexity of Type Inference Algorithm

• When Hindley/Milner type inference algorithm was developed, its complexity was unknown

• In 1989, Kanellakis, Mairson, and Mitchell proved that the problem was exponential-time complete

• Usually linear in practice though...
  – Running time is exponential in the depth of polymorphic declarations