Lesson 25

• Type inference
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.
  – Widely regarded as important language innovation.
  – Illustrative example of a flow-insensitive static analysis algorithm.
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,...
uHaskell

• Subset of Haskell to explain type inference.
  – Haskell and ML both have overloading
  – Will not cover 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

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

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

- Parse program text to construct parse tree.

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

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.
Step 3: Add Constraints

```latex
\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*}
\text{Fun} \quad f x = 2 + x
```
Step 4: Solve Constraints

\begin{align*}
  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} \\
  t_3 &= \text{Int} \\
  t_3 &= t_4 \rightarrow (\text{Int} \rightarrow \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} \\
  t_3 &= \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} \\
  t_2 &= \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
  t_3 &= \text{Int} \\
\end{align*}
Step 5: Determine type of declaration

$f x = 2 + x$

$\Rightarrow f :: \text{Int} \rightarrow \text{Int}$
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 (tail).
  - From form of parse tree: e.g., application and abstraction nodes.
- Solve constraints using unification
- Determine types of top-level declarations
Constraints from Application Nodes

- Function application (apply $f$ to $x$)
  - Type of $f$ ($t_0$ in figure) must be domain $\rightarrow$ 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 Abstraction

- Function declaration:
  - Type of $f$ (t_0 in figure) must domain $\rightarrow$ range
  - Domain is type of abstracted variable $x$ (t_1 in fig)
  - Range is type of function body $e$ (t_2 in fig)
  - Constraint: $t_0 = t_1 \rightarrow t_2$
Inferring Polymorphic Types

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

- Step 1:
  Build Parse Tree
Inferring Polymorphic Types

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

- Step 2:
  Assign type variables
Inferring Polymorphic Types

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

• Step 3:
  Generate constraints

\[ t_0 = t_1 \rightarrow t_4 \]
\[ t_1 = t_3 \rightarrow t_4 \]
\[ t_3 = \text{Int} \]
Inferring Polymorphic Types

• Example:

• Step 4:
  Solve constraints

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

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

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

Diagram:
- Fun
- \( f :: t_0 \)
- \( g :: t_1 \)
- \( (\emptyset) :: t_4 \)
- \( g :: t_1 \)
- \( 2 :: t_3 \)
Using Polymorphic Functions

• Function:

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

• Possible applications:

\[
\begin{align*}
\text{add} \ x &= 2 + x \\
> \ \text{add} &: \ \text{Int} \to \ \text{Int} \\
\text{f add} \\
> \ 4 &: \ \text{Int}
\end{align*}
\]

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

• Function:

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

• Incorrect use

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

• Type error:

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

• Example:
  \[ f (g, x) = g (g x) \]

• Step 1:
  Build Parse Tree
Another Example

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

• Step 2:
Assign type variables
Another Example

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

- Step 3:
Generate constraints
Another Example

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

- 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:
  \( f(g, x) = g(gx) \)
  
  > \( f :: (t_8 \rightarrow t_8, t_8) \rightarrow t_8 \)

• Step 5: Determine type of \( f \)

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

• Functions may have multiple clauses

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

• Type inference
  – Infer separate type for each clause
  – Combine by adding 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:

• Step 3: Solve Constraints

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

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

• Function with multiple clauses

\[
\begin{align*}
\text{append } ([], r) &= r \\
\text{append } (x:xs, r) &= x : \text{append } (xs, r)
\end{align*}
\]

• 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*

```haskell
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

```haskell
> 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*
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
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:

> reverse :: [t_1] -> [t_2]

• What does this type mean?

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

• Idea can be applied to other program properties
  – Discover properties of program using same kind of analysis
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