Lesson 29

• Type inference in ML / Haskell
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 do not consider overloading now

<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

\[ f \ x = 2 + x \]
\[ \Rightarrow f :: \text{Int} \to \text{Int} \]

• What is the type of \( f \)?

+ has type: \( \text{Int} \to \text{Int} \to \text{Int} \)

(with overloading would be \( \text{Num} \ a \Rightarrow a \to a \to a \))

2 has type: \( \text{Int} \)

Since we are applying + to \( x \) we need \( x :: \text{Int} \)

Therefore \( f \ x = 2 + x \) has type \( \text{Int} 	o \text{Int} \)
Step 1: Parse Program

- Parse program text to construct parse tree.

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

- 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.

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

• 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: $t_0 = t_1 \rightarrow t_2$
Step 3: Add Constraints

\[ f \cdot x = 2 + x \]

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

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

\[ f \cdot x = 2 + x \]

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

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

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

- Step 1:
  Build Parse Tree
Inferring Polymorphic Types

- Example:

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

- Step 2:
  Assign type variables
Inferring Polymorphic Types

• Example:

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

• Step 3:
Generate 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:

\[
\begin{align*}
  f \ast g &= g \ 2 \\
  > f &:: (\text{Int} \rightarrow t\_4) \rightarrow t\_4 \\
\end{align*}
\]

• Step 4:

Solve constraints

- \( t\_0 = t\_1 \rightarrow t\_4 \)
- \( t\_1 = t\_3 \rightarrow t\_4 \)
- \( t\_3 = \text{Int} \)

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

- Example:

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

- 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:

\[
\begin{align*}
\text{add } x &= 2 + x \\
\text{isEven } x &= \text{mod } (x, 2) == 0 \\
\end{align*}
\]

\[
\begin{align*}
> \text{add} &: \text{Int} \rightarrow \text{Int} \\
> \text{isEven} &: \text{Int} \rightarrow \text{Bool} \\
\end{align*}
\]

• Possible applications:

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

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

- Incorrect use
  \[
  \text{not } x = \text{if } x \text{ then True else False} \\
  > \text{not} :: \text{Bool} \rightarrow \text{Bool} \\
  f \ \text{not} \\
  > \text{Error: operator and operand don't agree} \\
  \text{operator domain: } \text{Int} \rightarrow a \\
  \text{operand: } \text{Bool} \rightarrow \text{Bool}
  \]

- Type error:
  cannot unify Bool → Bool and Int → 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:
  \[ f (g, x) = g (g x) \]
  \[ > f :: (t_8 \rightarrow t_8, \ t_8) \rightarrow t_8 \]

- Step 2:
Assign type variables
Another Example

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

• 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:

\[ f \circ (g, x) = g \circ (g \circ x) \]

> \[ f :: (t_8 \to t_8, t_8) \to t_8 \]

• Step 4:

Solve constraints

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

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

• Example:

\[ f \ (g, x) = g \ (g \ x) \]

\[ > f :: (t_8 \to t_8, t_8) \to t_8 \]

• Step 5:
Determine type of \( f \)

\[ t_0 = (t_8 \to t_8, t_8) \to t_8 \]

\[ t_0 = t_3 \to t_8 \]

\[ t_3 = (t_1, t_2) \]

\[ t_1 = t_7 \to t_8 \]

\[ t_1 = t_2 \to t_7 \]

26
Polymorphic Datatypes

• Functions may have multiple clauses

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

• 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:

  ```
  length (x:rest) = 1 + (length rest)
  ```

• Step 1: Build Parse Tree
Type Inference with Datatypes

- Example:

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

- 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

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

\[
\begin{align*}
  t_0 &= [t_1] \rightarrow \text{Int}
\end{align*}
\]
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*
  
  \[
  \text{map} \left( f, \left[ \right] \right) = \left[ \right]
  \]
  \[
  \text{map} \left( f, x : xs \right) = f \ x : \text{map} \left( f, xs \right)
  \]
  \[
  > \text{map} :: (t_1 \rightarrow t_2, \left[ t_1 \right]) \rightarrow \left[ t_2 \right]
  \]

- Functions may have many less general types
  
  \[
  > \text{map} :: (t_1 \rightarrow \text{Int}, \left[ t_1 \right]) \rightarrow \left[ \text{Int} \right]
  \]
  \[
  > \text{map} :: (\text{Bool} \rightarrow t_2, \left[ \text{Bool} \right]) \rightarrow \left[ t_2 \right]
  \]
  \[
  > \text{map} :: (\text{Char} \rightarrow \text{Int}, \left[ \text{Char} \right]) \rightarrow \left[ \text{Int} \right]
  \]

- Less general types are all instances of most general type, also called the *principal type*
Type Inference with overloading

• In presence of overloading (Type Classes), type inference infers a qualified type $Q \Rightarrow T$
  – $T$ is a Hindley Milner type, inferred as usual
  – $Q$ is set of type class predicates, called a constraint

• Consider the example function:

```example
example z xs =
  case xs of
    []     -> False
    (y:ys) -> y > z || (y==z && ys == [z])
```

  – Type $T$ is     $a \rightarrow [a] \rightarrow \text{Bool}$
  – Constraint $Q$ is $\{ \text{Ord } a, \text{Eq } a, \text{Eq } [a] \}$

Ord $a$ because  $y > z$
Eq $a$ because    $y == z$
Eq $[a]$ because  $ys == [z]$
Simplifying Type Constraints

• Constraint sets Q can be simplified:
  – Eliminate duplicates
    • \{\text{Eq a}, \text{Eq a}\} simplifies to \{\text{Eq a}\}
  – Use an \textit{instance declaration}
    • If we have instance Eq a => Eq [a],
    • then \{\text{Eq a}, \text{Eq [a]}\} simplifies to \{\text{Eq a}\}
  – Use a \textit{class declaration}
    • If we have class Eq a => Ord a where ..., 
    • then \{\text{Ord a}, \text{Eq a}\} simplifies to \{\text{Ord a}\}

• Applying these rules,
  – \{\text{Ord a}, \text{Eq a}, \text{Eq[a]}\} simplifies to \{\text{Ord a}\}
Type Inference with overloading

• Putting it all together:

```haskell
example z xs =
  case xs of
    []     -> False
    (y:ys) -> y > z || (y==z && ys ==[z])
```

- $T = a \rightarrow [a] \rightarrow \text{Bool}$
- $Q = \{\text{Ord } a, \text{Eq } a, \text{Eq } [a]\}$
- $Q$ simplifies to $\{\text{Ord } a\}$
- `example :: \{\text{Ord } a\} \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}`
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