SWEN430 - Compiler Engineering

Lecture 7 - Typing I

David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

Java Type Checking

The Java compiler must check:

- That primitive types are used correctly
- That reference types are used correctly
- That methods and fields exist with appropriate types
- That method overriding respects modifiers
- That generic types are used correctly
- That wildcard types are used correctly
- ...

```
class Test implements Inter <? extends Comparable> {
 double f(float f) {
  int x = (int) f;
  return f;
 }
 void g(Test x, Inter <? extends Comparable> y) {
  y = x; // up cast
  x = (Test) y; // down cast
 }
}
```

Java Binary Numeric Conversion

JLS 5.6.2 Binary Numeric Promotion:

When an operator applies binary numeric promotion to a pair of operands, each of which must denote a value that is convertible to a numeric type, the following rules apply, in order, using widening conversion (Section 5.1.2) to convert operands as necessary:

- *If any of the operands is of a reference type, unboxing conversion (Section 5.1.8) is performed. Then:*
- *If either operand is of type double, the other is converted to double.*
- *Otherwise, if either operand is of type float, the other is converted to float.*
- *Otherwise, if either operand is of type long, the other is converted to long.*
- *Otherwise, both operands are converted to type int.*

```
int i = 2;
float f = 1.0f;
double d = (i * f) * 2.0;
```

Recall, the λ-calculus!

$$
\begin{array}{lll}
\mathtt{t} & ::= & \\
 & | \quad \mathtt{x} & \text{(Variables)} \\
 & | \quad \mathtt{v} & \text{(Values)} \\
 & | \quad \mathtt{t\ t} & \text{(App)} \\
 & & \\
\mathtt{v} & ::= & \text{(Values)} \\
 & | \quad \lambda \mathtt{x.t} & \text{(Function)} \\
 & | \quad \mathtt{c} & \text{(Integer)}
\end{array}
$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{t_1\ t_2} \longrightarrow \mathtt{t_1'\ t_2}} \quad \text{(R-App1)}$$

$$\frac{\mathtt{t_2} \longrightarrow \mathtt{t_2'}}{\mathtt{v_1\ t_2} \longrightarrow \mathtt{v_1\ t_2'}} \quad \text{(R-App2)}$$

$$\frac{}{(\lambda \mathtt{x.t_1})\ \mathtt{v_1} \longrightarrow \mathtt{t_1}[\mathtt{x} \mapsto \mathtt{v_1}]} \quad \text{(R-App3)}$$

- A **simple** language — *syntax* and *semantics* on one slide!
- Useful starting point for **formalising** programming languages
- Term $\mathtt{t_1}[\mathtt{x} \mapsto \mathtt{t_2}]$ is $\mathtt{t_1}$ with all occurences of $\mathtt{x}$ replaced with $\mathtt{t_2}$

Type Checking

$$\Big(\big(\lambda x.(\lambda y.x\ y)\big)\ \big((\lambda z.z)\ 1\big)\Big)\ 1$$

- This program gets **stuck** before producing a *value*!
  - In languages like C, programs never get stuck ... they just crash!
  - In typesafe languages (e.g. Java), programs always raise errors before doing bad things
- Type checking checks our program will not get stuck
  - This is the approach used in **statically typed** languages
- Type checking suffers from **limitations of precision**
  - I.e. Correct programs can fail to type check

Simply Typed λ-Calculus - Syntax and Semantics

$$
\begin{array}{lll}
\mathtt{t} ::= & & \text{(Terms)} \\
\quad \mid \mathtt{x} & & \text{(Variables)} \\
\quad \mid \mathtt{v} & & \text{(Values)} \\
\quad \mid \mathtt{t\ t} & & \text{(Apps)} \\
 & & \\
\mathtt{v} ::= & & \text{(Values)} \\
\quad \mid \lambda \mathtt{x} : \mathtt{T}.\mathtt{t} & & \text{(Function)} \\
\quad \mid \mathtt{c} & & \text{(Integer)} \\
 & & \\
\mathtt{T} ::= & & \text{(Types)} \\
\quad \mid \mathtt{T} \to \mathtt{T} & & \text{(Fun type)} \\
\quad \mid \mathtt{int} & & \text{(Int type)}
\end{array}
$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \quad \text{(R-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \quad \text{(R-App2)}$$

$$\frac{}{(\lambda \mathtt{x} : \mathtt{T}.\mathtt{t}_2)\ \mathtt{v}_1 \longrightarrow \mathtt{t}_2[\mathtt{x} \mapsto \mathtt{v}_1]} \quad \text{(R-App3)}$$

Notation

$$\Gamma \vdash t_1 : T_1$$

- Γ is the **typing environment**
  - A set of pairs (v, T) mapping variables to types
  - Records the declared type for every variable
- $\Gamma \vdash t_1 : T_1$ is the **typing judgement**
  - In typing environment Γ, term t_1 can be shown to have type T_1

Type checking rules for λ-Calculus

$$\frac{}{\vdash \mathtt{n} : \mathtt{int}}\ \text{(T-Int)} \qquad \frac{\mathtt{x} : \mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}}\ \text{(T-Var)}$$

$$\frac{\Gamma \cup \{\mathtt{x} : \mathtt{T}_1\} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda \mathtt{x} : \mathtt{T}_1 . \mathtt{t}_2 : \mathtt{T}_1 \rightarrow \mathtt{T}_2}\ \text{(T-Fun)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \rightarrow \mathtt{T}_2 \quad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_1}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_2}\ \text{(T-App)}$$

Examples: 1 $(\lambda x : \mathit{int}.x)$ $(\lambda x : \mathit{int} \rightarrow \mathit{int}.x)$

Example Derivation

- A typed version of our **stuck** term:

$$\Big(\big(\lambda \mathtt{x} : \mathtt{int} \to \mathtt{int}.(\lambda \mathtt{y} : \mathtt{int}.\mathtt{x}\ \mathtt{y})\big)\big((\lambda \mathtt{z} : \mathtt{int}.\mathtt{z})\ 1\big)\Big)\ 1$$

- A *derivation tree* can be used to check the term's type:

$$\dfrac{\dfrac{\dfrac{\{\mathtt{z} : \mathtt{int}\} \vdash \mathtt{z} : \mathtt{int}}{\vdash (\lambda \mathtt{z} : \mathtt{int}.\mathtt{z}) : \mathtt{int} \to \mathtt{int}}\ \text{T-Fun} \quad \dfrac{}{\vdash 1 : \mathtt{int}}\ \text{T-Int}}{\vdash ((\lambda \mathtt{z} : \mathtt{int}.\mathtt{z})\ 1) : \mathtt{int}}\ \text{T-App}$$

$$\dfrac{\dfrac{\dfrac{\{\mathtt{x} : \mathtt{int} \to \mathtt{int}, \mathtt{y} : \mathtt{int}\} \vdash (\mathtt{x}\ \mathtt{y}) : \mathtt{int}}{\{\mathtt{x} : \mathtt{int} \to \mathtt{int}\} \vdash (\lambda \mathtt{y} : \mathtt{int}.(\mathtt{x}\ \mathtt{y})) : \mathtt{int}}\ \text{T-Fun}}{\vdash (\lambda \mathtt{x} : \mathtt{int} \to \mathtt{int}.(\lambda \mathtt{y} : \mathtt{int}.(\mathtt{x}\ \mathtt{y}))) : (\mathtt{int} \to \mathtt{int}) \to (\mathtt{int} \to \mathtt{int})}\ \text{T-Fun}$$

- No valid typing for $\mathtt{int}$ applied to $(\mathtt{int} \to \mathtt{int}) \to (\mathtt{int} \to \mathtt{int})$
  - So, our **stuck** term will not type check

Precision of Type Checking

Progress Theorem (Soundness)

A well-typed term $\mathtt{t}$ is not stuck (either $\mathtt{t}$ is a value or there exists some transition $\mathtt{t} \longrightarrow \mathtt{t}'$)

Preservation Theorem (Soundness)

If a well-typed term is evaluated one step, then the resulting term is also well typed (in fact, it has the same type)

Completeness

If evaluating term $\mathtt{t}$ does not get **stuck**, then there exists a valid typing of $\mathtt{t}$

- Our type system is sound, but not complete (this is impossible)
- There are valid programs which cannot be typed, such as:

$$\Big(\lambda \mathtt{f}.\mathtt{f}\ \mathtt{f}\ \mathtt{1}\Big)\ \lambda \mathtt{x}.\mathtt{x} \longrightarrow \Big((\lambda \mathtt{x}.\mathtt{x})\ (\lambda \mathtt{x}.\mathtt{x})\Big)\ \mathtt{1} \longrightarrow (\lambda \mathtt{x}.\mathtt{x})\ \mathtt{1} \longrightarrow \mathtt{1}$$

WHILE Compiler

$$\frac{\Gamma \vdash e_1 : \mathtt{T}[] \quad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash e_1[e_2] : \mathtt{T}} \text{ (T-ArrayAccess)}$$

```
public Type check(Expr.ArrayAccess expr, Map<String, Type> environment) {¶
    // Type source expression¶
    Type srcType = check(expr.getSource(), environment);¶
    // Type index expression¶
    Type indexType = check(expr.getIndex(), environment);¶
    // Check index has integer type¶
    checkInstanceOf(indexType, expr.getIndex(), Type.Int.class);¶
    // Check source has array type (of some kind)¶
    Type.Array t = checkInstanceOf(srcType, expr.getSource(), Type.Array.class);¶
    // Extract element type!¶
    return t.getElement();¶
}¶
```

- Typing rules for WHILE encoded in `TypeChecker`

Subtyping

```
Integer i = new Integer(1);
Float f = new Float(1.0);
Number n;
Object o;

n=i; // valid (because of subtyping)
n=f; // valid (because of subtyping)
n=o; // invalid (without cast)
```

- In Object-Oriented Languages (e.g. Java), *subtyping* is everywhere!
- In Java, subtyping and *subclassing* are the same thing
  - But, not in e.g. C++ (due to private inheritance)
- **Notation:** $\texttt{T}_1 \leq \texttt{T}_2$ means $\texttt{T}_1$ is a subtype of $\texttt{T}_2$

Simply Typed λ-Calculus with Subtyping ($\lambda_{\leq}$)

- Want subtype hierarchy where $\texttt{real} \leq \texttt{num}$ and $\texttt{int} \leq \texttt{num}$
- Want add operators:
  - $+ : (\texttt{int}, \texttt{int}) \rightarrow \texttt{int}$ (faster integer-only addition)
- Then, we could type "useful" functions such as:

$$\begin{array}{ll} \textbf{let} & \texttt{f} = (\lambda \texttt{x} : \texttt{num},\ \texttt{y} : \texttt{num} \rightarrow \texttt{int}.\ 1 + (\texttt{y}\ \texttt{x}))) \\ \textbf{in} & \texttt{f}\ \ 2.0\ \ (\lambda \texttt{z} : \texttt{num}.\ 1) \end{array}$$

- Note, "**let** $\texttt{x} = \texttt{e}_1$ **in** $\texttt{e}_2$" is syntactic sugar for $(\lambda \texttt{x}.\texttt{e}_2)\ \texttt{e}_1$

Syntax and Semantics for $\lambda_{\leq}$

$$
\begin{array}{lll}
\mathtt{t} ::= & & \text{(Terms)} \\
\quad \mid \mathtt{x} & & \text{(Variables)} \\
\quad \mid \mathtt{v} & & \text{(Values)} \\
\quad \mid \mathtt{t}\ \mathtt{t} & & \text{(Apps)} \\
\\
\mathtt{v} ::= & & \text{(Values)} \\
\quad \mid \lambda \mathtt{x} : \mathtt{T}.\mathtt{t} & & \text{(Function)} \\
\quad \mid \mathtt{c} & & \text{(Integer)} \\
\quad \mid \mathtt{c.c} & & \text{(Real)} \\
\\
\mathtt{T} ::= & & \text{(Types)} \\
\quad \mid \mathtt{T} \rightarrow \mathtt{T} & & \text{(Fun type)} \\
\quad \mid \mathtt{int} & & \text{(Int type)} \\
\quad \mid \mathtt{real} & & \text{(Real type)} \\
\quad \mid \mathtt{num} & & \text{(Num type)}
\end{array}
$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \quad \text{(App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \quad \text{(App2)}$$

$$\frac{}{(\lambda \mathtt{x} : \mathtt{T}.\mathtt{t}_2)\ \mathtt{v}_1 \longrightarrow \mathtt{t}_2[\mathtt{x} \mapsto \mathtt{v}_1]} \quad \text{(FunApp)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1 + \mathtt{t}_2 \longrightarrow \mathtt{t}_1' + \mathtt{t}_2} \quad \text{(Add1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1 + \mathtt{t}_2 \longrightarrow \mathtt{v}_1 + \mathtt{t}_2'} \quad \text{(Add2)}$$

$$\frac{}{\mathtt{v}_1 + \mathtt{v}_2 \longrightarrow \mathtt{v}_3} \quad \text{(Add3)}$$

Subtyping relation for $\lambda_{\leq}$

$$\frac{}{T_1 \leq T_1} \text{ (S-Refl)} \qquad \frac{T_1 \leq T_2 \quad T_2 \leq T_3}{T_1 \leq T_3} \text{ (S-Trans)}$$

$$\frac{}{\texttt{real} \leq \texttt{num}} \text{ (S-Real)} \qquad \frac{}{\texttt{int} \leq \texttt{num}} \text{ (S-Int)}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_3 \quad \Gamma \vdash t_2 : T_2 \quad T_2 \leq T_1}{\Gamma \vdash t_1\ t_2 : T_3} \text{ (T-App)}$$

- Subtyping relation is **reflexive** and **transitive**.
- T-App now supports **subtyping**.

Function Subtyping

$$\frac{\mathtt{T}_1 \geq \mathtt{T}_3 \quad \mathtt{T}_2 \leq \mathtt{T}_4}{\mathtt{T}_1 \rightarrow \mathtt{T}_2 \leq \mathtt{T}_3 \rightarrow \mathtt{T}_4} \text{ (S-Fun)}$$

- Subtyping of functions is **contravariant** in parameter position
  - Contravariant is when you can replace a type with a super type
  - Prevents this: $(\lambda \mathtt{x} : \mathtt{num} \rightarrow \mathtt{int}.\ \mathtt{x}\ \mathtt{1.0})\ (\lambda \mathtt{y} : \mathtt{int}.\ \mathtt{y} + \mathtt{y})$

- Subtyping of functions is **covariant** in result position
  - Covariant is when you can replace a type with a subtype
  - Prevents this: $(\lambda \mathtt{x} : \mathtt{num} \rightarrow \mathtt{int}.\ (\mathtt{x}\ \mathtt{1.0}) + \mathtt{1})\ (\lambda \mathtt{y} : \mathtt{num}.\ \mathtt{y})$