Compiling Techniques
Lecture 9: Semantic Analysis: Types

Christophe Dubach

9 October 2018
Table of contents

1. Type Systems
   - Specification
   - Type properties

2. Inference Rules
   - Inference Rules
   - Environments
   - Function Call

3. Implementation
   - Visitor implementation
Checking that identifiers are declared and used correctly is not the only thing that needs to be verified in the compiler. In most programming languages, expressions have a type. Therefore, we need to check that these types are correct and return an error message otherwise.
Examples: some typing rules of our Mini-C language

- The operands of `+` must be integers
- The operands of `==` must be compatible (int with int, char with char)
- The number of arguments passed to a function must be equal to the number of parameters
- ...
Definition: Strong/weak typing

A language is said to be **strongly typed** if the violation of a typing rule results in an error. A language is said to be **weakly typed** or not typed in other cases — in particular if the program behaviour becomes unspecified after an incorrect typing. Strong/weak typing is about **how strictly** types are distinguished (e.g. implicit conversion).
Definition: Strong/weak typing

A language is said to be **strongly typed** if the violation of a typing rule results in an error. A language is said to be **weakly typed** or not typed in other cases — in particular if the program behaviour becomes unspecified after an incorrect typing. Strong/weak typing is about **how strictly** types are distinguished (e.g. implicit conversion).

Definition: Static/dynamic typing

A language is said to be **statically typed** if there exists a type system that can detect incorrect programs before execution. A language is said to be **dynamically typed** in other cases. Static/dynamic typing is about **when** type information is available.
Warning
A strongly typed language does not necessarily imply static typing.

Examples
<table>
<thead>
<tr>
<th></th>
<th>strong</th>
<th>weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>Java</td>
<td>C/C++</td>
</tr>
<tr>
<td>dynamic</td>
<td>Python</td>
<td>JavaScript</td>
</tr>
</tbody>
</table>

- in Python: 'a'+1 will give a type error
- in JavaScript: 'a'+1 will produce 'a1'
We want to give an exact specification of the language.

- We will **formally** define this, using a mathematical notation.
- Programs who pass the type checking phase are **well-typed** since they corresponds to programs for which is it possible to give a **type** to each expression.

This mathematical description will fully specify the typing rules of our language.
Suppose that we have a small language expressing constants (integer literal), the + binary operation and the type int.

Example: language for arithmetic expressions

<table>
<thead>
<tr>
<th>Constants</th>
<th>i = a number (integer literal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expressions</td>
<td>e = i</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Types</td>
<td>T = int</td>
</tr>
</tbody>
</table>
An expression $e$ is of type $T$ iff:

- it's an expression of the form $i$ and $T = \text{int}$ or
- it's an expression of the form $e_1 + e_2$, where $e_1$ and $e_2$ are two expressions of type $\text{int}$ and $T = \text{int}$

To represent such a definition, it is convenient to use inference rules which in this context is called a typing rule:

**Typing rules**

\[
\begin{align*}
\text{IntLit} & \quad \vdash i : \text{int} \\
\text{BinOp} & \quad \vdash e_1 : \text{int}, \quad \vdash e_2 : \text{int} \\
& \quad \vdash e_1 + e_2 : \text{int}
\end{align*}
\]
Typing rules

\[
\text{INTLit} \quad \frac{}{\vdash i : \text{int}}
\]

\[
\text{BINOP} \quad \frac{\vdash e_1 : \text{int} \quad \vdash e_2 : \text{int}}{\vdash e_1 + e_2 : \text{int}}
\]

An inference rule is composed of:

- a horizontal line
- a name on the left or right of the line
- a list of premisses placed above the line
- a conclusion placed below the line

An inference rule where the list of premisses is empty is called an axiom.
An inference rule can be read bottom up:

Example

\[
\text{BINOP} \quad \vdash e_1 : \text{int} \quad \vdash e_2 : \text{int} \\
\text{\quad} \vdash e_1 + e_2 : \text{int}
\]

“To show that an expression of the form \( e_1 + e_2 \) has type \text{int}, we need to show that \( e_1 \) and \( e_2 \) have the type \text{int}”.

- To show that the conclusion of a rule holds, it is enough to prove that the premisses are correct.
- This process stops when we encounter an axiom.
Using the inference rule representation, it is possible to see whether an expression is well-typed.

Example: \((1+2)+3\)

\[
\begin{array}{cccc}
\text{BINOP} & \text{INTLIT} & \text{BINOP} & \text{INTLIT} \\
\quad & \vdash 1 : \text{int} & \quad & \vdash 2 : \text{int} \\
\quad & \vdash 1 + 2 : \text{int} & \quad & \vdash 3 : \text{int} \\
\quad & \vdash (1 + 2) + 3 : \text{int}
\end{array}
\]
Using the inference rule representation, it is possible to see whether an expression is well-typed.

**Example:** $(1+2)+3$

![Tree representation of the expression $(1+2)+3$]

Such a tree is called a derivation tree.

**Conclusion**

An expression $e$ has type $T$ iff there exist a derivation tree whose conclusion is $\vdash e : T$. 

Christophe Dubach

Compiling Techniques
Let's add identifiers to our language.

Example: language for arithmetic expressions

<table>
<thead>
<tr>
<th>Identifiers</th>
<th>x = a name (string literal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constants</td>
<td>i = a number (integer literal)</td>
</tr>
<tr>
<td>Expressions</td>
<td>e = i</td>
</tr>
<tr>
<td></td>
<td>\</td>
</tr>
<tr>
<td></td>
<td>\</td>
</tr>
<tr>
<td>Types</td>
<td>T = int</td>
</tr>
</tbody>
</table>

To determine if an expression such as \(x+1\) is well-typed, we need to have information about the type of \(x\).

We add an environment \(\Gamma\) to our typing rules which associates a type for each identifier. We now write \(\Gamma \vdash e : T\).
Environment

An typing environment $\Gamma$ is list of pairs of an identifier $x$ and a type $T$. We can add an inference rule to decide when an expression containing an identifier is well-typed:

$$\text{Ident} \quad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

Example: $x + 1$

In the environment $\Gamma = \{x: \text{int}\}$, it is possible to type check $x + 1$.
Environment

An typing environment $\Gamma$ is list of pairs of an identifier $x$ and a type $T$. We can add an inference rule to decide when an expression containing an identifier is well-typed:

\[
\text{IDENT} \quad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}
\]

Example: $x + 1$

In the environment $\Gamma = \{x : \text{int}\}$, it is possible to type check $x + 1$.

\[
\begin{align*}
\text{IDENT} & \quad \frac{x : T \in \Gamma}{\Gamma \vdash x : \text{int}} \\
\text{INTLIT} & \quad \frac{}{\Gamma \vdash 1 : \text{int}} \\
\text{BINOP} & \quad \frac{\Gamma \vdash x : \text{int} \quad \Gamma \vdash 1 : \text{int}}{\Gamma \vdash x + 1 : \text{int}}
\end{align*}
\]
We need to add a notation to talk about the type of the functions.

**Example: language for arithmetic expressions**

<table>
<thead>
<tr>
<th>Identifiers</th>
<th>$x = a$ name (string literal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constants</td>
<td>$i = a$ number (integer literal)</td>
</tr>
<tr>
<td>Expressions</td>
<td>$e = i$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td>Types</td>
<td>$T, U = int$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
</tbody>
</table>
Function call inference rule

\[
\text{FUNCALL}(f) \quad \frac{\Gamma \vdash f : \overline{U} \rightarrow T \quad \Gamma \vdash \overline{x} : \overline{U}}{\Gamma \vdash f(\overline{x}) : T}
\]

In plain English:

- the arguments $\overline{x}$ must be of types $\overline{U}$
- the function $f$ must be defined in the environment $\Gamma$ as a function taking parameters of types $\overline{U}$ and a return type $T$. 

Example: int foo(int, int)

FunCall(foo) $\Gamma$ $\vdash f : (\text{int}, \text{int}) \rightarrow \text{int}$$

$\Gamma$ $\vdash x_1 : \text{int}$

$\Gamma$ $\vdash x_2 : \text{int}$

$\Gamma$ $\vdash \text{foo}(x_1, x_2) : \text{int}$
Function call inference rule

\[
\text{\textsc{FunCall}}(f) \quad \frac{\Gamma \vdash f : \overline{U} \rightarrow T \quad \Gamma \vdash \overline{x} : \overline{U}}{\Gamma \vdash f(\overline{x}) : T}
\]

In plain English:

- the arguments \( \overline{x} \) must be of types \( \overline{U} \)
- the function \( f \) must be defined in the environment \( \Gamma \) as a function taking parameters of types \( \overline{U} \) and a return type \( T \).

Example: \( \text{int} \ \text{foo}(\text{int, int}) \)

\[
\text{\textsc{FunCall}}(\text{foo}) \quad \frac{\Gamma \vdash f : (\text{int, int}) \rightarrow \text{int} \quad \Gamma \vdash x_1 : \text{int} \quad \Gamma \vdash x_2 : \text{int}}{\Gamma \vdash \text{foo}(x_1, x_2) : \text{int}}
\]
\[ \text{BINOP}(+) \quad \vdash e_1 : \text{int} \quad \vdash e_2 : \text{int} \quad \vdash e_1 + e_2 : \text{int} \]

```java
public Type visitBinOp(BinOp bo) {
    // Visitor implementation
}
```
\[
\text{BINOP}(+) \quad \vdash e_1 : \text{int} \quad \vdash e_2 : \text{int} \\
\vdash e_1 + e_2 : \text{int}
\]

TypeChecker visitor : binary operation

```java
public Type visitBinOp(BinOp bo) {
    Type lhsT = bo.lhs.accept(this);
    Type rhsT = bo.lhs.accept(this);
    if (bo.op == ADD) {
        if (lhsT == Type.INT && rhsT == Type.INT) {
            bo.type = Type.INT; // set the type
            return Type.INT; // returns it
        } else {
            error();
        }
    } // . . .
}
```
\[ \text{BINOP}(+) \quad \vdash e_1 : \text{int} \quad \vdash e_2 : \text{int} \]

\[ \vdash e_1 + e_2 : \text{int} \]

TypeChecker visitor : binary operation

```java
public Type visitBinOp(BinOp bo) {
    Type lhsT = bo.lhs.accept(this);
    Type rhsT = bo.lhs.accept(this);
    if (bo.op == ADD) {
        // . . .
    }
}
```
\[
\text{BINOP}(+) \quad \vdash e_1 : \text{int} \quad \vdash e_2 : \text{int} \\
\quad \vdash e_1 + e_2 : \text{int}
\]

TypeChecker visitor : binary operation

```java
class TypeChecker {
    public Type visitBinOp(BinOp bo) {
        Type lhsT = bo.lhs.accept(this);
        Type rhsT = bo.lhs.accept(this);
        if (bo.op == ADD) {
            if (lhsT == Type.INT && rhsT == Type.INT) {
                bo.type = Type.INT; // set the type
                return Type.INT; // returns it
            } else {
                error();
            }
        } // . . .
    }
}
```
Type Systems
Inference Rules
Implementation

\[
\text{BINOP}(+) \quad \vdash e_1 : \text{int} \quad \vdash e_2 : \text{int} \\
\vdash e_1 + e_2 : \text{int}
\]

**TypeChecker visitor : binary operation**

```java
public Type visitBinOp(BinOp bo) {
    Type lhsT = bo.lhs.accept(this);
    Type rhsT = bo.lhs.accept(this);
    if (bo.op == ADD) {
        if (lhsT == Type.INT && rhsT == Type.INT) {
            bo.type = Type.INT; // set the type
        }
    }
    // . . .
}
```

Christophe Dubach  
Compiling Techniques
\[
\text{BINOP}(+) \quad \frac{\vdash e_1 : \text{int}}{} \quad \frac{\vdash e_2 : \text{int}}{} \quad \vdash e_1 + e_2 : \text{int}
\]

**TypeChecker visitor : binary operation**

```java
public Type visitBinOp(BinOp bo) {
    Type lhsT = bo.lhs.accept(this);
    Type rhsT = bo.lhs.accept(this);
    if (bo.op == ADD) {
        if (lhsT == Type.INT && rhsT == Type.INT) {
            bo.type = Type.INT; // set the type
            return Type.INT; // returns it
        }
    }
    // ...
}
```
\[
\text{BINOP}(+) \quad \vdash e_1 : \text{int} \quad \vdash e_2 : \text{int} \\
\vdash e_1 + e_2 : \text{int}
\]

TypeChecker visitor : binary operation

```java
class TypeChecker {
    public Type visitBinOp(BinOp bo) {
        Type lhsT = bo.lhs.accept(this);
        Type rhsT = bo.lhs.accept(this);
        if (bo.op == ADD) {
            if (lhsT == Type.INT && rhsT == Type.INT) {
                bo.type = Type.INT; // set the type
                return Type.INT; // returns it
            } else
                error();
        }
        // ...
    }
}
```

Christophe Dubach  Compiling Techniques
TypeChecker visitor: variables

```java
public Type visitVarDecl(VarDecl vd) {
    if (vd.type == VOID)
        error();
    return null;
}
```

Not just analysis!
The visitor does more than analysing the AST: it also remembers the result of the analysis directly in the AST node.
TypeChecker visitor: variables

```java
public Type visitVarDecl(VarDecl vd) {
    if (vd.type == VOID)
        error();
    return null;
}

public Type visitVarExp(Var v) {
    v.type = v.vd.type;
    return v.vd.type;
}
```
TypeChecker visitor: variables

```java
public Type visitVarDecl(VarDecl vd) {
    if (vd.type == VOID)
        error();
    return null;
}

public Type visitVarExp(Var v) {
    v.type = v.vd.type;
    return v.vd.type;
}
```

Not just analysis!
The visitor does more than analysing the AST: it also remembers the result of the analysis directly in the AST node.
Exercise: write the visit method for function call

```java
public Type visitFunCall(FunCall fc) {
    // ... }```

Function call inference rule

\[
\text{FUNCALL}(f) \quad \frac{\Gamma \vdash f : \overline{U} \rightarrow T \quad \Gamma \vdash x : \overline{U}}{\Gamma \vdash f(x) : T}
\]
Conclusion

- Typing rules can be formally defined using inference rules.
- We saw how to implement them with a visitor

Next lecture:
- An introduction to Assembly