Compiling Techniques

Lecture 9: Semantic Analysis: Types

Christophe Dubach

11 October 2016

Coursework announcement 1

Grammar has been simplified

```
stmt        ::=   lexp "=" exp ";"  |  ...
```

has been replaced by

```
stmt        ::=   exp "=" exp ";"  |  ...
```

Coursework announcement 2

Dealing with arrayaccess and fieldaccess

You can use the following trick to remove left recursion:

$A \rightarrow A\alpha_1 | A\alpha_2 | \ldots | A\alpha_m | \beta_1 | \beta_2 | \ldots | \beta_n$ where
$first(\beta_i) \cap first(A) = \varnothing$ and $\varepsilon \notin first(\alpha_i)$

can be rewritten into:

$A \rightarrow \beta_1 A' | \beta_2 A' | \ldots | \beta_n A'$
$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_m A' | \varepsilon$

Table of contents

What are types used for?

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
- ...

Typing properties

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).

Typing properties

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 types 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

	strong	**weak**
static	Java	C/C++
dynamic	Python	JavaScript

- in Python: `'a'+1` will give a type error
- in JavaScript: `'a'+1` will produce `'a1'`

Goal

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

Constants	i	=	*a number (integer literal)*
Expressions	e	=	i
		|	$e_1 + e_2$
Types	T	=	**int**

An expression `e` is of type `T` iff:

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

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

Typing rules

$$\textsc{IntLit}\ \frac{}{\vdash i : \textbf{int}} \qquad \textsc{BinOp}\ \frac{\vdash e_1 : \textbf{int} \qquad \vdash e_2 : \textbf{int}}{\vdash e_1 + e_2 : \textbf{int}}$$

Typing rules

$$\textsc{IntLit}\ \frac{}{\vdash i : \textbf{int}} \qquad \textsc{BinOp}\ \frac{\vdash e_1 : \textbf{int} \qquad \vdash e_2 : \textbf{int}}{\vdash e_1 + e_2 : \textbf{int}}$$

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

$$\textsc{BinOp}\ \frac{\vdash e_1 : \textbf{int} \qquad \vdash e_2 : \textbf{int}}{\vdash e_1 + e_2 : \textbf{int}}$$

"To show that an expression of the form $e_1 + e_2$ has type **int**, we need to show that $e1$ and $e2$ have the type **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 possible to see whether an expression is well-typed.

Example: (1+2)+3

$$\textsc{BinOp}\ \dfrac{\textsc{BinOp}\ \dfrac{\textsc{IntLit}\ \dfrac{}{\vdash 1 : \textbf{int}} \quad \textsc{IntLit}\ \dfrac{}{\vdash 2 : \textbf{int}}}{\vdash 1 + 2 : \textbf{int}} \quad \textsc{IntLit}\ \dfrac{}{\vdash 3 : \textbf{int}}}{\vdash (1 + 2) + 3 : \textbf{int}}$$

Using the inference rule representation, it possible to see whether an expression is well-typed.

Example: (1+2)+3

$$\textsc{BinOp}\ \dfrac{\textsc{BinOp}\ \dfrac{\textsc{IntLit}\ \dfrac{}{\vdash 1 : \textbf{int}} \quad \textsc{IntLit}\ \dfrac{}{\vdash 2 : \textbf{int}}}{\vdash 1 + 2 : \textbf{int}} \quad \textsc{IntLit}\ \dfrac{}{\vdash 3 : \textbf{int}}}{\vdash (1 + 2) + 3 : \textbf{int}}$$

Such a tree is called a derivation tree.

Conclusion

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

Identifiers

Let's add identifiers to our language.

Example: language for arithmetic expressions

Identifiers	x	=	*a name (string literal)*
Constants	i	=	*a number (integer literal)*
Expressions	e	=	i
		|	$e_1 + e_2$
		|	x
Types	T	=	**int**

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 Γ to our typing rules which associates a type for each identifier. We now write $\Gamma \vdash e : T$.

Environment

An typing environment Γ 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:

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

Environment

An typing environment Γ 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:

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

Example: $x + 1$

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

$$\textsc{BinOp}\ \frac{\textsc{Ident}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : \textbf{int}} \qquad \textsc{IntLit}\ \frac{}{\Gamma \vdash 1 : \textbf{int}}}{\Gamma \vdash x + 1 : \textbf{int}}$$

Function call

We need to add a notation to talk about the type of the functions.

Example: language for arithmetic expressions

Identifiers	x	=	*a name (string literal)*
Constants	i	=	*a number (integer literal)*
Expressions	e	=	i
		|	$e_1 + e_2$
		|	x
Types	T, U	=	**int**
		|	$\overline{U} \rightarrow T$

Function call inference rule

$$\textsc{FunCall}(\mathrm{f})\ \frac{\Gamma \vdash f : \overline{U} \rightarrow T \qquad \Gamma \vdash \overline{x} : \overline{U}}{\Gamma \vdash f(\overline{x}) : T}$$

In plain English:

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

Function call inference rule

$$\textsc{FunCall}(\text{f})\ \frac{\Gamma \vdash f : \overline{U} \rightarrow T \qquad \Gamma \vdash \overline{x} : \overline{U}}{\Gamma \vdash f(\overline{x}) : T}$$

In plain English:

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

Example: int foo(int, int)

$$\textsc{FunCall}(\text{foo})\ \frac{\Gamma \vdash f : (\textbf{int}, \textbf{int}) \rightarrow \textbf{int} \qquad \Gamma \vdash x1 : \textbf{int} \qquad \Gamma \vdash x2 : \textbf{int}}{\Gamma \vdash foo(x_1, x_2) : \textbf{int}}$$

$$\textsc{BinOp}(+)\ \frac{\vdash e_1 : \textbf{int} \qquad \vdash e_2 : \textbf{int}}{\vdash e_1 + e_2 : \textbf{int}}$$

TypeChecker visitor : binary operation

```
public Type visitBinOp(BinOp bo) {
```

$$\textsc{BinOp}(+)\ \frac{\vdash e_1 : \textbf{int} \qquad \vdash e_2 : \textbf{int}}{\vdash e_1 + e_2 : \textbf{int}}$$

TypeChecker visitor : binary operation

```
public Type visitBinOp(BinOp bo) {
  Type lhsT = bo.lhs.accept(this);
  Type rhsT = bo.lhs.accept(this);
```

$$\textsc{BinOp}(+)\ \frac{\vdash e_1 : \textbf{int} \qquad \vdash e_2 : \textbf{int}}{\vdash e_1 + e_2 : \textbf{int}}$$

TypeChecker visitor : binary operation

```
public Type visitBinOp(BinOp bo) {
  Type lhsT = bo.lhs.accept(this);
  Type rhsT = bo.lhs.accept(this);
  if (bo.op == ADD) {
```

$$\textsc{BinOp}(+)\ \frac{\vdash e_1 : \textbf{int} \qquad \vdash e_2 : \textbf{int}}{\vdash e_1 + e_2 : \textbf{int}}$$

TypeChecker visitor : binary operation

```
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) {
```

$$\textsc{BinOp}(+)\ \frac{\vdash e_1 : \textbf{int} \qquad \vdash e_2 : \textbf{int}}{\vdash e_1 + e_2 : \textbf{int}}$$

TypeChecker visitor : binary operation

```
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
```

$$\textsc{BinOp}(+)\ \frac{\vdash e_1 : \textbf{int} \qquad \vdash e_2 : \textbf{int}}{\vdash e_1 + e_2 : \textbf{int}}$$

TypeChecker visitor : binary operation

```
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
```

$$\textsc{BinOp}(+)\ \frac{\vdash e_1 : \textbf{int} \qquad \vdash e_2 : \textbf{int}}{\vdash e_1 + e_2 : \textbf{int}}$$

TypeChecker visitor : binary operation

```
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();
  }
  // ...
}
```

TypeChecker visitor: variables

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

TypeChecker visitor: variables

```
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

```
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

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

Function call inference rule

$$\textsc{FunCall}(\mathrm{f})\ \frac{\Gamma \vdash f : \overline{U} \rightarrow T \qquad \Gamma \vdash \overline{x} : \overline{U}}{\Gamma \vdash f(\overline{x}) : T}$$

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