# Compiler Design

# Lecture 10: Semantic Analysis: part II Types

Christophe Dubach Winter 2025

Timestamp: 2025/02/06 17:24:00

Type Systems Specification Type properties

Inference Rules

Inference Rules

Environments

Function Call

Implementation with Pattern-Matching

Type Systems

**Type Systems** 

Specification

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.

Types are here to ensure that expressions are compatible with one another to guarantee some level of correctness.

#### MY NEW LANGUAGE 15 GREAT, BUT IT HAS A FELL QUIRKS REGARDING TYPE:

[1]>	2+*2*
[2]>	*2* + []
=>	°[2]*
[3]	(2/O)
	(0, (0) · 2
141 >	(2/0)+2 N-P
~~~	
< 10	(1/2)
- /	E a al a
[6] >	[1,2,3]+Z
- / 5-7 \	
= >	17215 17215
- / [6] \	2/2-(3/2-1/2))
= >	NaN.000000000000000
[9] >	RANGE(" ")
= >	(' " ' " lo " " " " " ' ' ' ' ' )
[i0] >	+2
=>	12
[11] >	2+2
=>	DONE
[14] >	RANGE(1,5)
=>	(1,4,3,4,5)
[13] >	FL00R(10.5)
= >	
=>	
= >	1 10 5
=>	110.5

### Examples: 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

• . . .

**Type Systems** 

Type properties

# Typing properties

### 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).

### 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

# 🛕 A strongly typed language does not imply static typing. 🛕

Language examp	les			
		strong	weak	
	static	Java	C/C++	
	dynamic	Python	JavaScript	

### Java (static/strong)

class A {}
class B {}
B b = new B();
A a = (A) b;
// compile-time error

### Python (dynamic/strong)

1+'a' # run-time error

# C (static/weak)

```
int * p1;
char ** p2;
p1 = (int*) p2;
// no error
```

### JavaScript (dynamic/weak)

```
3 + '6'; // '36'

3 * '6'; // 18

num = 11;

num.toUpperCase();

// run-time error
```

#### Weak dynamic typing: the worst of the worst!

### JavaScript

num = 11; num.toUpperCase(); // run-time error 3 + '6'; // '36' 3 \* '6'; // 18 // no error



source: http://gunshowcomic.com/648

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.

# **Inference Rules**

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	е	=	i	
			<i>e</i> <sub>1</sub> + <i>e</i> <sub>2</sub>	
Types	Т	=	int	

### We want to define a type *judgement* (*a.k.a.* statement):

 $\vdash e: \tau$ 

In english: I can "conclude" that expression e has type  $\tau$ .

**Inference Rules** 

**Inference Rules** 

An expression e is of type T iff:

- it's an expression of the form *i* and *T* = **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 = int

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

Typing rules

INTLIT 
$$\longrightarrow i:$$
 int BINOP  $\frac{\vdash e_1:$  int  $\vdash e_2:$  int  $\vdash e_1 + e_2:$  int

# Typing rules

An inference rule is composed of:

- $\cdot$  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

$$\mathsf{BINOP} \frac{\vdash e_1 : \mathsf{int} \quad \vdash e_2 : \mathsf{int}}{\vdash e_1 + e_2 : \mathsf{int}}$$

"To show that an expression of the form  $e_1 + e_2$  has type **int**, we need to show that  $e_1$  and  $e_2$  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.



Such a tree is called a derivation tree.

#### Conclusion

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

# **Inference Rules**

Environments

# Identifiers

Let's add identifiers to our language.

Example:	languag	ge	for arithmetic expressions
Identifier	S X	=	a name (string literal)
Constants	i	=	a number (integer literal)
Expression	s e	=	i
			<i>e</i> <sub>1</sub> + <i>e</i> <sub>2</sub>
			Х
Types	Т	=	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  $\Gamma$  to our typing rules which associates a type for each identifier.

Our type judgement are now written as:  $\Gamma \vdash e : \tau$ . In english: given  $\Gamma$ , I can conclude *e* has type  $\tau$ . A typing environment  $\Gamma$  is list of pairs of an identifier x and a type T. It can be implemented in two ways in the compiler:

- As a symbol table;
- Or directly encoded in the AST nodes: *e.g.* VarExpr node has a reference to the declaration (filled in during *Name Analysis*)

We can add an inference rule to decide when an expression containing an identifier is well-typed:

 $\mathsf{IDENT} \ \frac{X:T \in \mathsf{\Gamma}}{\mathsf{\Gamma} \vdash x:T}$ 



**Inference Rules** 

Function Call

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

Example: language for arithmetic expressions					
Identifiers	Х	// string			
Constants	i	// integer			
Expressions	e =	i			
		<i>e</i> <sub>1</sub> + <i>e</i> <sub>2</sub>			
		Х			
Types	⊤,U =	int			
		$(U_1,\ldots,U_n) \to T$			

where  $(U_1, \ldots, U_n) \rightarrow T$  represents a function type.

### Function call inference rule

$$\mathsf{FUNCALL}(\mathsf{X}) \xrightarrow{\Gamma \vdash \mathsf{X} : (U_1, \dots, U_n) \to T} \xrightarrow{\Gamma \vdash e_1 : U_1} \dots \xrightarrow{\Gamma \vdash e_n : U_n} \Gamma \vdash f(e_1, \dots, e_n) : T$$

In plain English:

- each argument  $x_i$  must be of type  $U_i$
- the function f is defined in the environment  $\Gamma$  as a function taking parameters of types  $U_1, \ldots, U_n$  and a return type T.

### Example: int foo(int, int)

$$\mathsf{FUNCALL}(\mathsf{foo}) \frac{\Gamma \vdash \mathsf{foo}: (\mathsf{int}, \mathsf{int}) \to \mathsf{int} \quad \Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash \mathsf{foo}(e_1, e_2) : \mathsf{int}}$$

Implementation with Pattern-Matching

### TypeChecker

```
class TypeChecker {
  Type visit (ASTnode node) {
    return switch(node) {
    ...
    }
  }
}
```

The visit method returns the type inferred for the AST node (if any).

$$\mathsf{BINOP}(+) \frac{\vdash e_1 : \mathsf{int} \quad \vdash e_2 : \mathsf{int}}{\vdash e_1 + e_2 : \mathsf{int}}$$

#### TypeChecker : binary operation

```
case BinOp bo \rightarrow {
 Type lhsT = bo.lhs.visit();
 Type rhsT = bo.rhs.visit();
  if (bo.op == ADD) {
    if (lhsT == Type.INT && rhsT == Type.INT) {
      bo.type = Type.INT; // sets the type
      yield Type.INT; // returns it
    } else
      error():
      yield Type.INVALID;
```

#### TypeChecker: variables

```
case VarDecl vd → {
    if (vd.type == VOID)
    error();
    yield Type.NONE;
}
case Var v → {
    v.type = v.vd.type;
    yield v.vd.type;
}
```

### Not just analysis!

The type checker does more than analysing the AST: it also remembers the result of the analysis directly in the AST node.

$$\mathsf{FUNCALL}(\mathsf{X}) \xrightarrow{\Gamma \vdash \mathsf{X} : (U_1, \dots, U_n) \to T \qquad \Gamma \vdash e_1 : U_1 \qquad \dots \qquad \Gamma \vdash e_n : U_n}{\Gamma \vdash \mathsf{X}(e_1, \dots, e_n) : T}$$

# Exercise: write the case for function call

case FunCall fc  $\rightarrow$  {

- Typing rules can be formally defined using inference rules.
- $\cdot$  We saw how to implement them with a pattern-matching

Next lecture:

• An introduction to MIPS Assembly