Type Checking

Md. Khorshed Alam
CSE, NDUB

Introduction

- To do type checking a compiler needs to assign a type expression to each component of the source program.
- The compiler must then determine that these type expressions conform to a collection of logical rules that is called the type system for the source language

Importance

- Type checking has the potential for catching errors in programs.
- In principle, any check can be done dynamically, if the target code carries the type of an element along with the value of the element
- A **sound** type system eliminates the need for dynamic checking for type errors, because it allows us to determine statically that these errors cannot occur when the target program runs.
- An implementation of a language is **strongly typed** if a compiler guarantees that the programs it accepts will run without type errors.

Importance

- Besides their use for compiling, ideas from type checking have been used to improve the security of systems that allow software modules to be imported and executed.
- Java programs compile into machine-independent bytecodes that include detailed type information about the operations in the bytecodes.
- Imported code is checked before it is allowed to execute, to guard against both inadvertent errors and malicious misbehavior.

- **Must check**: Syntactic and Semantic errors
- **Dynamic Checking**: during execution of target program

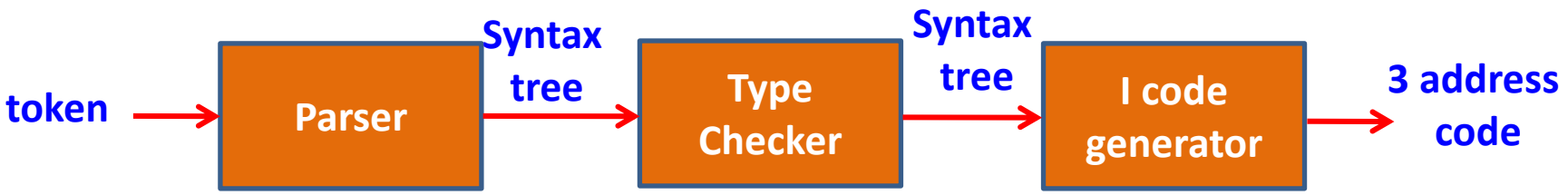
- **Static Checking**: before execution

❑ **Examples:**

1. **Type checks:** incompatible operand
2. **Flow-of-control checks:** enclosing braces
3. **Uniqueness checks:** identifier must declared uniquely
4. **Name-related checks:** same name beginning & end of a construct

Function of Type Checker

- A type checker verifies that the type of a construct matches that expected by its context.
- Type checking uses logical rules to reason about the behavior of a program at run time.



- Specifically, it ensures that the types of the operands match the type expected by an operator.
- For example, the && operator in Java expects its two operands to be Booleans; the result is also of type Boolean.

Type Systems

- The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, & the rules for assigning types to language constructs

➢ If both operands of arithmetic operators of addition, subtraction, & multiplication are of type integer, then the result is of type integer

➢ The result of the unary & operator is a pointer to the object referred to by the operand.

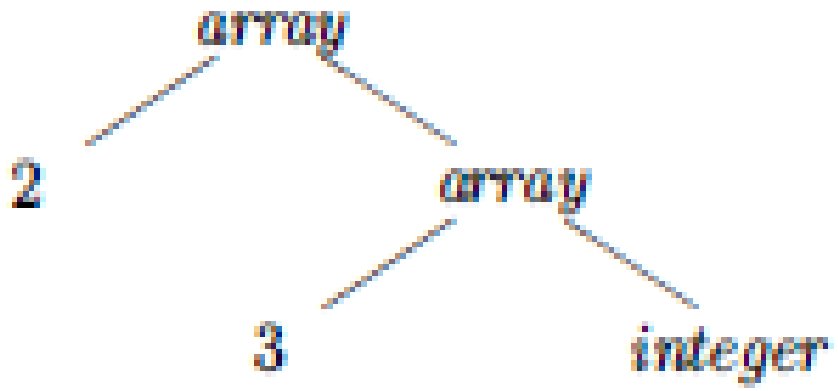
➢ If the type of the operand is '---', the type of the result is 'pointer to---'

Type Expressions

- Types have structure, which we shall represent using type expressions: a type expression is **either** a *basic type* or is formed by applying an operator called a *type constructor* to a type expression.
- The sets of basic types and constructors depend on the language to be checked.

Example 6.8: The array type `int[2][3]` can be read as "array of 2 arrays of 3 integers each" and written as a type expression *array*(2, *array*(3, *integer*)). This type is represented by the tree in Fig. 6.14. The operator *array* takes two parameters, a number and a type. □

type expression: array(2, array(3, integer)



Type expression for ***int [2] [3]***

Definition

1. A basic type is a type expression. Typical basic types for a language include *boolean, char, integer, float*, and *void*; the latter denotes "the absence of a value."

2. A type name is a type expression.
3. A type expression can be formed by applying the *array* type constructor to a number and a type expression.
4. A record is a data structure with named fields. A type expression can be formed by applying the ***record*** type constructor to the field names & their types.
5. A type expression can be formed by using the type constructor $\rightarrow$ for function types. We write **s $\rightarrow$t** for "function from type s to type t"

6. If **s** & **t** are type expressions, then their Cartesian product **s x t** is a type expression.
7. Type expressions may contain variables whose values are type expressions

A convenient way to represent a type expression is to use a graph/Dag

Interior nodes: type constructor
Leaves: basic types, type name, type variables

Type Equivalence

- When are two type expressions equivalent?

- Many type-checking rules have the form, "***if two type expressions are equal then return a certain type else error***."
- Potential ambiguities arise when names are given to type expressions and the names are then used in subsequent type expressions.
- The key issue is whether a name in a type expression stands for itself or whether it is an abbreviation for another type expression.

- When type expressions are represented by graphs, two types are structurally equivalent if and only if one of the following conditions is true:

➢ They are the same basic type.

➢ They are formed by applying the same constructor to structurally equivalent types

➢ One is a type name that denotes the other.

Operations Within Expressions

- The syntax-directed definition builds up the three-address code for an assignment statement ***S*** using attribute code for ***S*** and attributes ***addr*** and code for an expression **E**.
- Attributes $S_{.code}$ and $E_{.code}$ denote the three-address code for S and E, respectively.
- Attribute $E_{.addr}$ denotes the address that will hold the value of E.

PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \ \|\|$ $gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \ \|\| \ E_2.code \ \|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \ \|\|$ $gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \textbf{id}$	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

Consider the last production, $E \rightarrow \textbf{id}$, in the syntax-directed definition in Fig. 6.19. When an expression is a single identifier, say x, then x itself holds the value of the expression. The semantic rules for this production define $E.addr$ to point to the symbol-table entry for this instance of **id**. Let *top* denote the current symbol table. Function *top.get* retrieves the entry when it is applied to the string representation **id**.*lexeme* of this instance of **id**. $E.code$ is set to the empty string.

When $E \rightarrow (E_1)$, the translation of E is the same as that of the subexpression E_1. Hence, $E.addr$ equals $E_1.addr$, and $E.code$ equals $E_1.code$.

The operators + and unary - in Fig. 6.19 are representative of the operators in a typical language. The semantic rules for $E \rightarrow E_1 + E_2$, generate code to compute the value of E from the values of E_1 and E_2. Values are computed into newly generated temporary names. If E_1 is computed into $E_1.addr$ and E_2 into $E_2.addr$, then $E_1 + E_2$ translates into $t = E_1.addr + E_2.addr$, where t is a new temporary name. $E.addr$ is set to t. A sequence of distinct temporary names $t_1, t_2, \ldots$ is created by successively executing **new** $Temp()$.

When we translate the production $E \rightarrow E_1 + E_2$, the semantic rules in Fig. 6.19 build up $E.code$ by concatenating $E_1.code$, $E_2.code$, and an instruction that adds the values of E_1 and E_2. The instruction puts the result of the addition into a new temporary name for E, denoted by $E.addr$.

The translation of $E \rightarrow -E_1$ is similar. The rules create a new temporary for E and generate an instruction to perform the unary minus operation.

Finally, the production $S \rightarrow \mathbf{id} = E;$ generates instructions that assign the value of expression E to the identifier **id**. The semantic rule for this production uses function *top.get* to determine the address of the identifier represented by **id**, as in the rules for $E \rightarrow \mathbf{id}$. *S.code* consists of the instructions to compute the value of E into an address given by *E.addr*, followed by an assignment to the address *top.get*(**id**.*lexeme*) for this instance of **id**.

Example 6.11: The syntax-directed definition in Fig. 6.19 translates the assignment statement `a = b + - c ;` into the three-address code sequence

$$
\begin{aligned}
t_1 &= \texttt{minus c} \\
t_2 &= \texttt{b} + t_1 \\
\texttt{a} &= t_2
\end{aligned}
$$

Rules for Type Checking

- Type checking can take on two forms: ***synthesis and inference***.
- Type synthesis builds up the type of an expression from the types of its subexpressions. It requires names to be declared before they are used.
- The type of $\mathbf{E_1 + E_2}$ is defined in terms of the *types of E_1 and E_2*
- A typical rule for type synthesis has the form

> if f has type $s \to t$ and x has type s,
> then expression $f(x)$ has type t

- Here *f* and *x* denote expressions, and $\mathbf{s \to t}$ denotes a function from *s* to *t*.
- This rule for functions with one argument carries over to functions with several arguments. The rule can be adapted for $\mathbf{E_1 + E_2}$ by viewing it as a function application **add(E_1. E_2)**

- ***Type inference*** determines the type of a language construct from the way it is used.
- Let null be a function that tests whether a list is empty. Then, from the usage *null(x)* , we can tell that x must be a list.
- The type of the elements of x is not known; all we know is that x must be a list of elements of some type that is presently unknown.

- Variables representing type expressions allow us to talk about unknown types.
- We shall use Greek letters α, β,..... for type variables in type expressions.
- A typical rule for type inference has the form

> **if** $f(x)$ is an expression,
> **then** for some α and β, f has type $\alpha \rightarrow \beta$ **and** x has type α

- The rules for checking statements are similar to the rules for expressions.
- For example, we treat the conditional statement

" **if (E) S**;" as if it were the application of a function *if* to **E** and **S**.

- Let the special type *void* denote the absence of a value.
- Then function *if* expects to be applied to a *boolean* and a *void*;
- the result of the application is a *void*.

Type Conversions

- Consider expressions like **x + i**, where **x** is of type *float* and **i** is of type *integer*
- Since the representation of integers & floating-point numbers is different within a computer & different machine instructions are used for operations on integers & floats, the compiler may need to convert one of the operands of + to ensure that both operands are of the *same type when the addition occurs.*

- Suppose that *integers are converted to floats* when necessary, using a unary operator (*float*)
- For example, the *integer 2 is converted to a float* in the code for the expression 2 * 3 . 14

```
t1 = (float) 2
t2 = t1 * 3.14
```

Semantic Rules for Conversion

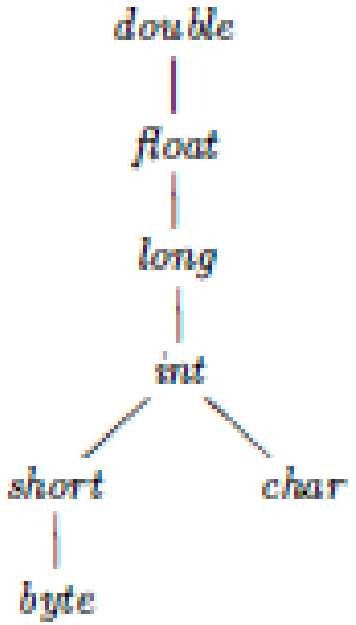
- We introduce attribute $E_{.type}$, whose value is either integer or float.
- The rule associated with $\mathbf{E \rightarrow E_1 + E_2}$ builds on the pseudocode

```
if ( E1.type = integer and E2.type = integer )  E.type = integer;
else if ( E1.type = float and E2.type = integer )  ...
...
```

Type conversion rules vary from language to language

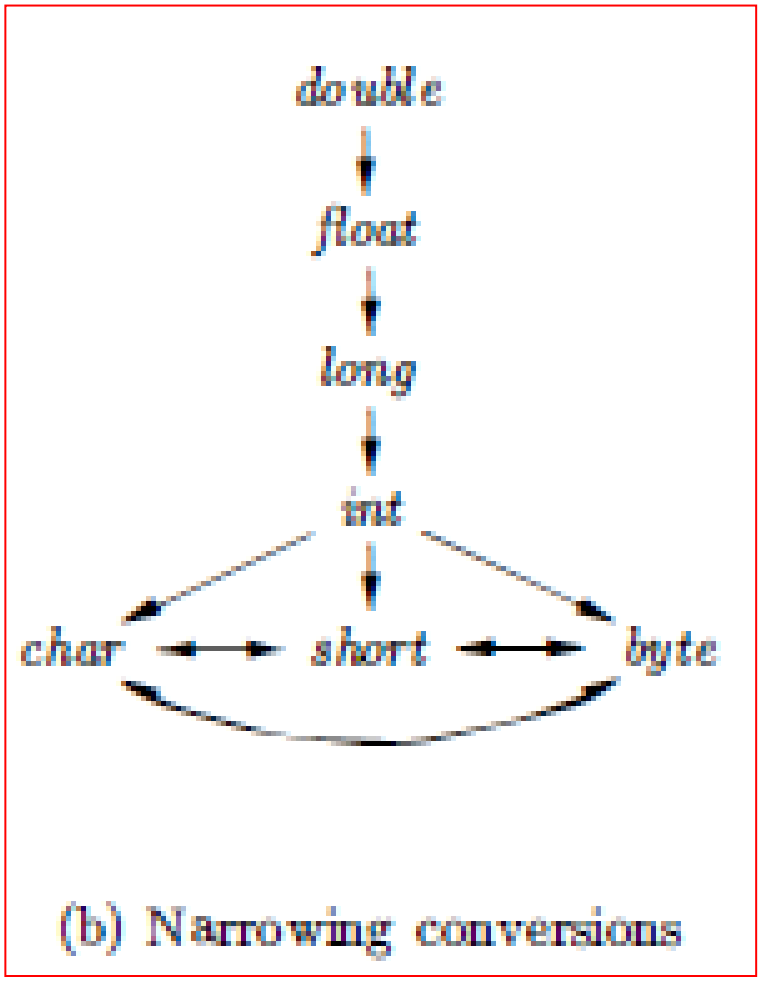
- The rules for Java
- ***widening*** conversions, which are intended to preserve information,
- ***narrowing*** conversions, which can lose information

- The widening rules are given by the hierarchy :any type lower in the hierarchy can be widened to a higher type.
- Thus, a ***char*** can be widened to an ***int*** or to a ***float***, but a ***char*** cannot be widened to a short.



(a) Widening conversions

- The narrowing rules : a type ***s*** can be narrowed to a type ***t*** if there is a path from ***s*** to ***t***.
- Note that ***char***, ***short***, & ***byte*** are pair-wise convertible to each other.



(b) Narrowing conversions

Implicit & Explicit Conversions

- **Implicit**: Conversion from one type to another is said to be implicit if *it is done automatically by the compiler*.
  - Implicit type conversions, also called ***coercions***, are limited in many languages to widening conversions.
- **Explicit**: Conversion is said to be explicit if *the programmer must write something to cause the conversion*.
  - Explicit conversions are also called **casts**.

Semantic action for checking $E \rightarrow E_1 + E_2$ uses two functions:

1. $max(t_1, t_2)$ takes two types t_1 and t_2 and returns the maximum (or least upper bound) of the two types in the widening hierarchy. It declares an error if either t_1 or t_2 is not in the hierarchy; e.g., if either type is an array or a pointer type.

2. $widen(a, t, w)$ generates type conversions if needed to widen an address a of type t into a value of type w. It returns a itself if t and w are the same type. Otherwise, it generates an instruction to do the conversion and place the result in a temporary t, which is returned as the result. Pseudocode for *widen*, assuming that the only types are *integer* and *float*, appears in Fig. 6.26.

```
Addr widen(Addr a, Type t, Type w)
        if ( t = w ) return a;
        else if ( t = integer and w = float ) {
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else error;
}
```

Figure 6.26: Pseudocode for function *widen*

Introducing Type Conversions into Expression

- The semantic action for $E \rightarrow E_1 + E_2$ illustrates how type conversions can be added for translating expressions.

$$
\begin{aligned}
E \rightarrow E_1 + E_2 \quad \{\ & E.type = max(E_1.type, E_2.type); \\
& a_1 = widen(E_1.addr, E_1.type, E.type); \\
& a_2 = widen(E_2.addr, E_2.type, E.type); \\
& E.addr = \textbf{new}\ Temp(); \\
& gen(E.addr\ '='\ a_1\ '+'\ a_2);\ \}
\end{aligned}
$$

- In the semantic action, *temporary variable* a_1 is either $E_{1.addr}$, if the type of E_1 does not need to be converted to the type of E , or a new temporary variable returned by widen if this conversion is necessary.
- Similarly, a_2 is either $E_{2.addr}$ or a new temporary holding the type-converted value of E_2
- Neither conversion is needed if both types are integer or both are float.
- In general, however, we could find that the only way to add values of two different types is to convert them both to a third type.

$$
\begin{aligned}
E \rightarrow E_1 + E_2 \quad \{\ & E.type = max(E_1.type, E_2.type); \\
& a_1 = widen(E_1.addr, E_1.type, E.type); \\
& a_2 = widen(E_2.addr, E_2.type, E.type); \\
& E.addr = \textbf{new}\ Temp(); \\
& gen(E.addr\ '='\ a_1\ '+'\ a_2);\ \}
\end{aligned}
$$