COMPILER CONSTRUCTION (Intermediate Code Generation) **Fall 2019**

Lecture 24: Type-Checking

Instructor: K.R. Chowdhary *: Professor of CS*

24.1 Types and Declarations

The applications of data types can be grouped under *checking* and *translation*:

- *Type checking.* It 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 Boolean; the result is also of type Boolean.
- *Translation Applications.* From the type of a name, a compiler can determine the storage that will be needed for that name at run time. Type information is also needed to calculate the address denoted by an array reference, to insert explicit type conversions, and to choose the right version of an arithmetic operator, among other things.

In the following we will examine types and storage layout for names declared within a procedure or a class; the actual storage for a procedure call or an object is allocated at run time, when the procedure is called or the object is created. As we examine local declarations at compile time, we can, however, use *relative-addresses*[1].

24.1.1 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 24.1 *Type Array.*

An array type `int`[2] can be read as $array(2, integer)$. Thus, the array type `int`[2][3] (e.g., *int* $a[2][3]$) can be read as "array of 2 arrays of, 3 integers each" and written as a type expression $\texttt{array}(2, \texttt{array}(3, \texttt{integer}))$. This type is represented by a tree shown in Fig. 24.1. The operator *array* always takes two parameters, a number and a type.

[1]A relative address of a name or a component of a data structure is an offset from the start of a data area.

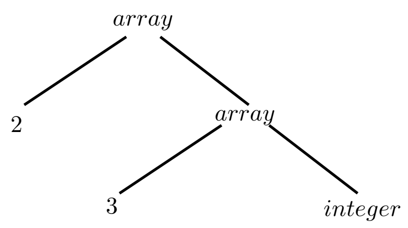

Figure 24.1: Type expression for `int`[2][3].

□

We shall use the following definition of type expressions:

- 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."
- A type name is also a type expression.
- A type expression can be formed by applying the *array* type constructor to a number and a type expression (see Fig. 24.1).
- 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 and their types. Record types can be implemented by applying the constructor record to a symbol table containing entries for the fields.
- 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." Function types will be useful when we discuss type checking later on.
- If s and t are type expressions, then their Cartesian product, $s \times t$, is a type expression. Products are introduced for completeness; they can be used to represent a list or tuple of types (e.g., for function parameters). We assume that the operator "$\times$" is left associative it has higher precedence than "$\rightarrow$".
- Type expressions may contain variables whose values are type expressions.

As we noted above, the type declaration is done using simplified grammar that declares just one name at a time. The declarations with list of names can be handles using, e.g., following grammar.

$$\begin{aligned}
D &\rightarrow T\ \mathbf{id}; D \mid \varepsilon \\
T &\rightarrow B\ C \mid \mathbf{record}\ '\{'\ D\ '\}' \\
B &\rightarrow \mathbf{int} \mid \mathbf{float} \\
C &\rightarrow \varepsilon \mid [\mathbf{num}]\ C
\end{aligned} \tag{24.1}$$

The fragment of the above grammar deals with basic and array type. The nonterminal D generates a sequence of declarations. The nonterminal T generates basic, array, or record types. The nonterminal B generates one of the basic types **int** and **float**. The nontermal C

(for component) generates string of zero or integers, each integer surrounded by brackets. An array type consists of a basic type specified by B, followed by array components specified by nonterminal C. A record type (the second production of T) is a sequence of declarations for the fields of the record, all surrounded by curly braces.

A convenient way to represent a type expression is to use a graph. The value-number method can be adapted to construct a DAG for a type expression, with interior nodes for type constructors and leaves for basic types, type names, and type variables.

24.1.2 Type equivalence

Many type-checking rules have the form, "if two type expressions are equal then return certain error." Potential ambiguities arise when names are given to two type expressions, and the names are used in subsequent expressions. When type expressions are represented by graphs, they are *structurally equivalent* if and only if one of the following conditions hold:

- 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.

24.2 Storage layout for names

From the type of a name, we can determine the amount of storage that will be needed for the name at run time. At compile time we can assign to each name a *relative address.* The types and relative-addresses are saved in the symbol table entry for name. Data of various length, such as strings, or data whose size cannot be determined until run time, such as *dynamic arrays*, is handled by reserving a known fixed amount of storage for a pointer to the data.

The translation scheme (SDT) given below in table 24.1 computes the type and their widths for basic and array types. The SDT uses synthesized attribute *type* and *width* for each non-terminal and two variables t and w, to pass type and width information from B node in a parse-tree to the node for the production $C \to \varepsilon$. In the SDD, t and w would be inherited attribute for C.

Table 24.1: Computing types and their widths

$T \to B$	$\{t{=}B.type;\ w{=}B.width;\}$
$\quad C$	
$B \to$ **int**	$\{B.type{=}integer;\ B.width{=}4;\}$
$B \to$ **float**	$\{B.type{=}float;\ B.width{=}8;\}$
$C \to \varepsilon$	$\{C.type{=}t;\ C.width{=}w;\}$
$C \to$ [**num**] C_1	$\{array(\textbf{num}.value,\ C_1.type);$
	$C.width{=}\textbf{num}.value \times C_1.width;\}$

24.3 Type Checking

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.

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. Otherwise, the language is called *weakly-typed*. C is weakly-typed, while Java is strongly-typed.

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 misbehaviour.

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 $E_1 + E_2$ is defined in terms of the types of E_1 and E_2. A typical rule for type synthesis has the form,

$$\begin{array}{l}\textbf{if } f \text{ has type } s \to t \text{ and } x \text{ has type } s,\\ \textbf{then } \text{expression } f(x) \text{ has type } t\end{array} \tag{24.2}$$

Here, f and x denote expressions, and $s \to t$ denotes a function from type s to type t. This rule for functions with one argument carries over to functions with several arguments. The rule (24.2) can be adapted for $E_1 + E_2$ by viewing it as a function application $add(E_1, E_2)$. Other examples are *mult*, *bigger*, *sqroot*.

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. In other words, type inference provides a type of an element due to the way it is used. For example, in C, when *char* is used in add operation, it is treated as integer, like, in $int\ a;\ x = x +' a';$.

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

$$\begin{array}{l}\textbf{if } f(x) \text{ is an expression,}\\ \textbf{then } \text{for some } \alpha \text{ and } \beta, f \text{ has type } \alpha \to \beta \text{ and } x \text{ has type } \alpha\end{array} \tag{24.3}$$

Type inference is needed for languages like ML, which check types, but do not require names to be declared.

In this section, we consider type checking of expressions. The rules for checking statements are similar to those 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.*

24.4 Control Flow

The translation of statements such as *if-else*-statements and *while*-statements are tied to the translation of Boolean expressions. In programming languages, Boolean expressions are often used to,

1. *Alter the flow of control.* Boolean expressions are used as conditional expressions in statements that alter the flow of control. The value of such boolean expressions is implicit in a position reached in a program. For example, in **if**(E) S, the expression E must be true if statement S is reached.
2. *Compute logical values.* A Boolean expression can represent *true* or *false* as it value. Such Boolean expressions can be evaluated in analogy to arithmetic expressions using three-address instructions with logical operators.

The intended use of Boolean expressions is determined by its syntactic context. For example, an expression following the keyword **if** is used to alter the flow of control, while an expression on the right side of an assignment is used to denote a logical value. Such syntactic contexts can be specified in a number of ways: we may use two different nonterminals, use inherited attributes, or set a flag during parsing. Alternatively we may build a syntax tree and invoke different procedures for the two different uses of Boolean expressions.

24.4.1 Boolean Expressions

Boolean expressions are composed of the Boolean operators (which we denote &&, ||, and !, in the C convention for the operators AND, OR, and NOT, respectively) applied to elements that are Boolean variables or relational expressions. Relational expressions are of the form E_1 **rel** E_2, where E_1 and E_2 are arithmetic expressions. In this section, we consider Boolean expressions (B) generated by the following grammar:

$$B \to B || B \mid B \&\& B \mid \; ! \; B \mid (B) \mid E \ \mathbf{rel} \ E \mid \mathbf{true} \mid \mathbf{false}$$

We use the attribute **rel**.*op* to indicate which of the six comparison operators $<, <=, >, >=, =, !$ represented by **rel**. As is customary, we assume that || and && are left-associative, and that || has lowest precedence, then &&, then !.

Given the expression B_1 || B_2, if we determine that B_1 is true, then we can conclude that the entire expression is true without having to evaluate B_2. Similarly, given B_1 && B_2, if B_1 is false, then the entire expression is false.

The semantic definition of the programming language determines whether all parts of a boolean expression must be evaluated. If the language definition permits (or requires) portions of a boolean expression to go unevaluated, then the compiler can optimize the evaluation of boolean expressions by computing only enough of an expression to determine its value. Thus, in an expression such as $B_1 \mid\mid B_2$, neither B_1 nor B_2 is necessarily evaluated fully. If either B_1 or B_2 is an expression with side effects (e.g., it contains a function that changes a global variable), then an unexpected answer may be obtained.

24.4.2 Short-Circuit Code

It is name given to a code with implicit or explicit jump statement. In short-circuit code, the boolean operators &&, ||, and ! translate into jumps. The operators themselves do not appear in the code; instead, the value of a boolean expression is represented by a position in the code sequence.

Example 24.2 *Convert* $if(x < 100 \mid\mid x > 200 \&\& \ x! = y) \ x = 0;$ *into intermediate code.*

The statement might be translated into the code below. In this translation, the Boolean expression is true if control reaches label L_2. If the expression is false, control goes immediately to label L_1, skipping L_2 and the assignment $x = 0$.

```
     if x < 100 goto L2
     if False x > 200 goto L1
     if False x != y goto L1
L2:  x = 0
L1:  ..
```

□

24.5 Review Questions

1. Explain the meaning of declaration $array(2, array(3, integer))$.
2. What is strongly typed language? What are its advantages and disadvantages.
3. What are the advantages and disadvantages of weakly-typed languages.
4. Does a Java program perform more rigorous type checking than a C program? Explain it, beyond the Yes/No.

24.6 Exercises

1. What are the two forms of type-checking? Explain the difference.
2. Give the type expression for $int[5][6][7]$.
3. Classify the following languages as strongly-typed and weakly-typed: C, C++, Java, Fortran, Prolog, LISP, Python.

4. A dynamic type-checking in a program is useful against malicious code. Justify this statement.
5. Write a note on type-checking for Java.
6. How a run-time checking of a program leads to more secured program? Suggest your own idea with logical justification.

References

[1] Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho, Monica S. Lam, et al., Sep 10, 2006.

[2] Compiler design in C (Prentice-Hall software series) by Allen I Holub, Jan 1, 1990.

[3] Engineering a Compiler, by Keith D. Cooper and Linda Torczon, Morgan Kaufmann Publishers, 2004.

[4] Tools for Large-scale Parser Development, Proceedings of the COLING-2000 Workshop on Efficiency In Large-Scale Parsing Systems, 2000, pp. 54-54, `http://dl.acm.org/citation.cfm?id=2387596.2387604`.