Formal Languages and Compilers
Lecture VIII—Semantic Analysis: Type Checking & Symbol Table

Alessandro Artale

Free University of Bozen-Bolzano
Faculty of Computer Science – POS Building, Room: 2.03
artale@inf.unibz.it
http://www.inf.unibz.it/∼artale/

Formal Languages and Compilers — BSc course

2020/21 – Second Semester

Summary of Lecture VIII

- Type Checking
  - Type System
  - Specifying a Type Checker
  - Type Conversion
- Symbol Table
  - Scope Rules and Symbol Tables
  - Translation Schemes for building Symbol Tables

Type Checking: Intro

- A compiler must check that the program follows the *Type Rules* of the language.
- Information about *Data Types* is maintained and computed by the compiler.
- The *Type Checker* is a module of a compiler devoted to type checking tasks.
- **Examples of Tasks.**
  1. The operator mod is defined only if the operands are integers;
  2. Indexing is allowed only on an array and the index must be an integer;
  3. A function must have a precise number of arguments and the parameters must have a correct type;
  4. etc...

Type Checking: Intro (Cont.)

- Type Checking may be either *static* or *dynamic*: The one done at compile time is static.
- In languages like Java, C and Pascal type checking is primarily static and is used to check the correctness of a program before its execution.
- Static type checking is also useful to determine the amount of memory needed to store variables.
- The design of a Type Checker depends on the syntactic structure of language constructs, the *Type Expressions* of the language, and the rules for assigning types to constructs.

Summary

- Type Checking
  - Type System
  - Specifying a Type Checker
  - Type Conversion
- Symbol Table
  - Scope Rules and Symbol Tables
  - Translation Schemes for building Symbol Tables

Type Expressions

- A Type Expression denotes the type of a language construct.
- A type expression is either a *Basic Type* or is built applying *Type Constructors* to other types.
  1. A *Basic Type* is a type expression (int, real, boolean, char). The basic type void represents the empty set and allows statements to be checked;
  2. Type expressions can be associated with a name: *Type Names* are type expressions;

Type Expressions (Cont.)

3. A *Type Constructor* applied to type expressions is a type expression. Type constructors inlude:
   1. *Array*. If T is a type expression, then $array(I, T)$ is a type expression denoting an array with elements of type T and index range in I—e.g., **array**[1..10] of int == array(1..10,int).
   2. *Product*. If T_1 and T_2 are type expressions, then their Cartesian Product $T_1 \times T_2$ is a type expression.
   3. *Record*. Similar to Product but with names for different fields (used to access the components of a record). Example of a C record type:

      struct
      { DOUBLE real-field;
        INT integer-field; }

      is of type expression

      $$record((\text{real-field} \times \text{DOUBLE}) \times (\text{integer-field} \times \text{INT}))$$

Type Expressions (Cont.)

4. *Pointer*. If T is a type expression, then **pointer**(T) is the type expression "pointer to an object of type T";
5. *Function*. If D is the domain and R the range of the function then we denote its type by the type expression: $D : R$.
   The **mod** operator has type, $int \times int : int$. The Pascal function:

   function f(a, b: char): int

   has type:

   $$char \times char : int$$

Type System

- Type System: Collection of (semantic) rules for assigning type expressions to the various part of a program.
- We will show how syntax directed definition can be used to specify Type Systems.
- A *type checker* implements a type system.
- **Definition.** A language is *strongly typed* if its compiler can guarantee that the program it accepts will execute without type errors.

Summary

- Type Checking
  - Type System
  - Specifying a Type Checker
  - Type Conversion
- Symbol Table
  - Scope Rules and Symbol Tables
  - Translation Schemes for building Symbol Tables

Specification of a Type Checker

- We specify a type checker for a simple language where identifiers have an associated type.
- Grammar for Declarations and Expressions:

$$P \to D; E$$
$$D \to D; D \mid \texttt{id} : T$$
$$T \to \texttt{char} \mid \texttt{int} \mid \texttt{array}[\texttt{num}] \; \texttt{of} \; T \mid \; \uparrow T$$
$$E \to \texttt{num} \mid \texttt{id} \mid E \; \texttt{mod} \; E \mid E[E] \mid E\uparrow$$

Specification of a Type Checker (Cont.)

- The syntax directed definition for associating a type to an *Identifier* is:

Production	Semantic Rule
$D \rightarrow \texttt{id} : T$	$addtype(\texttt{id}.entry, T.type)$
$T \rightarrow \texttt{char}$	$T.type := char$
$T \rightarrow \texttt{int}$	$T.type := int$
$T \rightarrow \uparrow T_1$	$T.type := pointer(T_1.type)$
$T \rightarrow \texttt{array[num] of } T_1$	$T.type := array(1..\texttt{num}.val, T_1.type)$

- All the attributes are synthesized.
- Since $P \rightarrow D; E$, all the identifiers will have their types saved in the symbol table before type checking an expression E.
- id.entry is the pointer to entry in the Symbol Table storing the identifier.

Specification of a Type Checker (Cont.)

- The syntax directed definition for associating a type to an *Expression* if `num` stands for an integer number, is:

Production	Semantic Rule
$E \rightarrow \texttt{num}$	$E.type := int$
$E \rightarrow \texttt{id}$	$E.type := lookup(\texttt{id}.entry)$
$E \rightarrow E_1 \texttt{ mod } E_2$	$E.type :=$ `if` $E_1.type = int$ `and` $E_2.type = int$ `then` int `else` $type\_error$
$E \rightarrow E_1[E_2]$	$E.type :=$ `if` $E_2.type = int$ `and` $E_1.type = array(i,t)$ `then` t `else` $type\_error$
$E \rightarrow E_1\uparrow$	$E.type :=$ `if` $E_1.type = pointer(t)$ `then` t `else` $type\_error$

Specification of a Type Checker (Cont.)

- The syntax directed definition for associating a type to a *Statement* is:

Production	Semantic Rule
$S \to \texttt{id} := E$	$S.type :=$ if id.$type = E.type$ then $void$ else $type\_error$
$S \to$ if E then S_1	$S.type :=$ if $E.type = boolean$ then $S_1.type$ else $type\_error$
$S \to$ while E do S_1	$S.type :=$ if $E.type = boolean$ then $S_1.type$ else $type\_error$
$S \to S_1; S_2$	$S.type :=$ if $S_1.type = void$ and $S_2.type = void$ then $void$ else $type\_error$

- The type expression for a statement is either *void* or *type_error*.
- **Final Remark.** For languages with type names or (even worst) allowing sub-typing we need to define when two types are *equivalent*.

Type Checker for Functions

Production	Semantic Rule
$Fun \rightarrow \texttt{fun id}(D) : T ; B$	$addtype(\texttt{id}.entry, D.type{:}T.type)$
$D \rightarrow \texttt{id} : T$	$addtype(\texttt{id}.entry, T.type); D.type := T.type$
$D \rightarrow D_1 ; D_2$	$D.type := D_1.type \times D_2.type$
$B \rightarrow \{S\}$	
$S \rightarrow \texttt{id}(EList)$	$S.type :=$ `if` $lookup(\texttt{id}.entry) = t_1 : t_2$ `and` $EList.type = t_1$ `then` t_2 `else` $type\_error$
$EList \rightarrow E$	$EList.type := E.type$
$EList \rightarrow EList , E$	$EList.type := EList_1.type \times E.type$

Summary

- Type Checking
  - Type System
  - Specifying a Type Checker
  - Type Conversion
- Symbol Table
  - Scope Rules and Symbol Tables
  - Translation Schemes for building Symbol Tables

Type Conversion

- **Example.** What's the type of "`x + y`" if:
  1. `x` is of type `real`;
  2. `y` is of type `int`;
  3. Different machine instructions are used for operations on reals and integers.
- Depending on the language, specific conversion rules are adopted by compilers to convert the type of one of the operand of +
  - The type checker in a compiler can insert these conversion operators into the intermediate code.
  - Such an implicit type conversion is called *Coercion*.

Type Coercion in Expressions

- The syntax directed definition for coercion from integer to real for a generic arithmetic operation op is:

PRODUCTION	SEMANTIC RULE
$E \rightarrow \texttt{num}$	$E.type := int$
$E \rightarrow \texttt{realnum}$	$E.type := real$
$E \rightarrow \texttt{id}$	$E.type := lookup(\texttt{id}.entry)$
$E \rightarrow E_1 \texttt{ op } E_2$	$E.type :=$ if $E_1.type = int$ and $E_2.type = int$ then int else if $E_1.type = int$ and $E_2.type = real$ then $real$ else if $E_1.type = real$ and $E_2.type = int$ then $real$ else if $E_1.type = real$ and $E_2.type = real$ then $real$ else $type\_error$

Summary

- Type Checking
  - Type System
  - Specifying a Type Checker
  - Type Conversion
- Symbol Table
  - Scope Rules and Symbol Tables
  - Translation Schemes for building Symbol Tables

Symbol Table

- The Symbol Table is the major inherited attribute and the major data structure as well.
- Symbol Tables store information about the name, type, scope and allocation size.
- Symbol Table must maintain efficiency against insertion and lookup.
- Dynamic data structures must be used to implement a symbol table: Linked Lists and Hash Tables are the most used.
  - Each entry has the form of a record with a field for each peace of information.

Relative Address in the Symbol Table

- Relative Address. Is a storage allocation information consisting of an offset from a base (usually zero) address: The Loader will be responsible for the run-time storage.
- The following translation scheme computes such address using a global variable called *offset*.
- The function `enter` adds into the Symbol Table: name, type and offset for each identifier.

$P \to \{offset := 0\}\ D$

$D \to D; D$

$D \to \texttt{id} : T \quad \{enter(\texttt{id}.name, T.type, offset);\ offset := offset + T.width\}$

$T \to \texttt{int} \quad \{T.type := int; T.width := 4\}$

$T \to \texttt{real} \quad \{T.type := real; T.width := 8\}$

$T \to \texttt{array}[\texttt{num}]\ \texttt{of}\ T_1 \quad \{T.type := array(\texttt{num}.val, T_1.type);\ T.width := \texttt{num}.val * T_1.width\}$

$T \to \uparrow T_1 \quad \{T.type := pointer(T_1.type); T.width := 4\}$

Relative Address (Cont.)

- The global variable *offset* keeps track of the next available address.
  - Before the first declaration, *offset* is set to 0;
  - As each new identifier is seen it is entered in the symbol table and *offset* is incremented.
- *type* and *width* are synthesized attributes for non-terminal T.

Summary

- Type Checking
  - Type System
  - Specifying a Type Checker
  - Type Conversion
- Symbol Table
  - Scope Rules and Symbol Tables
  - Translation Schemes for building Symbol Tables

Symbol Tables and Scope Rules

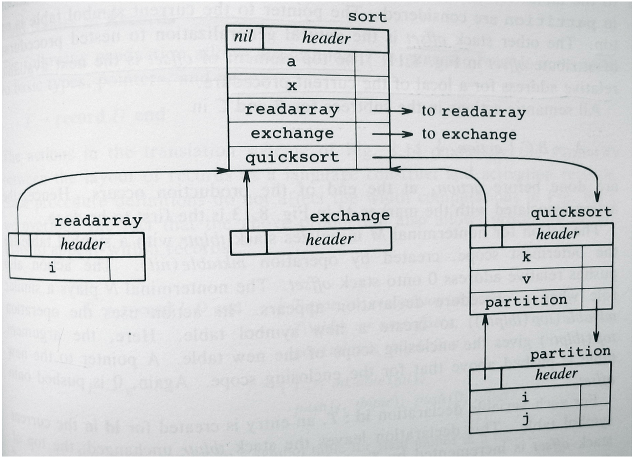
- A **Block** in a programming language is any set of language constructs that can contain declarations.
- A language is **Block Structured** if
  1. Blocks can be *nested* inside other blocks, and
  2. The *Scope* of declarations in a block is limited to that block and the blocks contained in that block.
- **Most Closely Nested Rule.** Given several different declarations for the same identifier, the declaration that applies is the one in the most closely nested block.

Symbol Tables and Scope Rules (Cont.)

- To implement symbol tables complying with nested scopes
  1. The *insert* operation into the symbol table must not overwrite previous declarations;
  2. The *lookup* operation into the symbol table must always refer to the most close block rule;
  3. The *delete* operation must delete only the most recent declarations.
- The symbol table behaves in a stack-like manner.

Symbol Tables and Scope Rules (Cont.)

- One possible solution to implement a symbol table under nested scope is to maintain separate symbol tables for each scope.
- Tables must be linked both from inner to outer scope, and from outer to inner scope.



Summary

- Type Checking
  - Type System
  - Specifying a Type Checker
  - Type Conversion
- Symbol Table
  - Scope Rules and Symbol Tables
  - Translation Schemes for building Symbol Tables

Lexical- Vs. Syntactic-Time Construction

1. Information is first entered into a symbol table by the lexical analyzer only if the programming language does not allow for different declarations for the same identifier (scope).
2. If scoping is allowed, the lexical analyzer will only return the name of the identifier together with the token:
   - The identifier is inserted into the symbol table when the syntactic role played by the identifier is discovered.

Keeping Track of Scope Information

- Let's consider the case of *Nested Procedures*: When a nested procedure is seen processing of declarations in the enclosing procedure is suspended.
- To keep track of nesting a stack is maintained.
- We associate a new symbol table for each procedure:
  - When we need to *enter* a new identifier into a symbol table we need to specify which symbol table to use.

Keeping Track of Scope Information (Cont.)

- We are now able to provide the translation scheme for processing declarations in nested procedure.
  - We use markers non-terminals, M, N, to rewrite productions with embedded rules.

$P \rightarrow M\ D$	$\{\texttt{addwidth}(\texttt{top}(tblptr), \texttt{top}(offset));$ $\texttt{pop}(tblptr); \texttt{pop}(offset)\}$
$M \rightarrow \epsilon$	$\{t := \texttt{mktable}(nil);$ $\texttt{push}(t, tblptr); \texttt{push}(0, offset)\}$
$D \rightarrow D_s; D \mid D_s$	
$D_s \rightarrow \texttt{proc id}; N D; S$	$\{t := \texttt{top}(tblptr); \texttt{addwidth}(t, \texttt{top}(offset));$ $\texttt{pop}(tblptr); \texttt{pop}(offset);$ $\texttt{enterproc}(\texttt{top}(tblptr), \texttt{id}.name, t)\}$
$D_s \rightarrow \texttt{id} : T$	$\{\texttt{enter}(\texttt{top}(tblptr), \texttt{id}.name, T.type, \texttt{top}(offset));$ $\texttt{top}(offset) := \texttt{top}(offset) + T.width\}$
$N \rightarrow \epsilon$	$\{t := \texttt{mktable}(\texttt{top}(tblptr));$ $\texttt{push}(t, tblptr); \texttt{push}(0, offset)\}$

Keeping Track of Scope Information (Cont.)

The semantic rules make use of the following procedures and stack variables:

1. *mktable(previous)* creates a new symbol table and returns its pointer. The argument *previous* is the pointer to the enclosing procedure.
2. The stack *tblptr* holds pointers to symbol tables of the enclosing procedures.
3. The stack *offset* keeps track of the relative address w.r.t. a given nesting level.
4. *enter(table,name,type,offset)* creates a new entry for the identifier *name* in the symbol table pointed to by *table*, specifying its *type* and *offset*.
5. *addwidth(table,width)* records the cumulative *width* of all the entries in *table* in the header of the symbol table.
6. *enterproc(table,name,newtable)* creates a new entry for procedure *name* in the symbol table pointed to by *table*. The argument *newtable* points to the symbol table for this procedure *name*.

Summary of Lecture VIII

- Type Checking
  - Type System
  - Specifying a Type Checker
  - Type Conversion
- Symbol Table
  - Scope Rules and Symbol Tables
  - Translation Schemes for building Symbol Tables