Chapter 7

Type Checking

7.1 Introduction

- The compiler must perform static checking (checking done at compile time). This ensures that certain types of programming errors will be detected and reported.

- Some examples of static checks are:
  
  - Type checks. A compiler should report an error if an operator is applied to an incompatible operand.
  
  - Flow-of-control checks. Statements that cause flow of control to leave a construct must have some place to which to transfer flow of control. For example, branching to nonexistent labels.
  
  - Uniqueness checks. Objects should be defined only once. This is true in many languages.
  
  - Name-related checks. Sometimes, the same name must appear two or more times. For examples, in Ada the name of a block must appear both at the beginning of the block and at the end. In Modula2, think about module names and procedure names.

- We will talk about type checkers. They verify that the type of a construct matches that expected by its context.

- Type information gathered by a type checker may be needed when code is generated. For example, arithmetic operators may be different at the machine level for different types of operands (real and integer).

- A symbol that can represent different operations in different contexts is said to be overloaded.

- Overloading may be accompanied by coercion of types, where a compiler supplies an operator to convert an operand into the type expected by the context.

- A distinct notion from overloading is that of polymorphism. The body of a polymorphic function can be executed with arguments of several types. We will also discuss this a little later.
7.2 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, and the rules for assigning types to language constructs.
- Talk a little about C in this respect.
- In most languages, types are either basic or constructed. Basic types are the atomic types with no internal structure as far as the programmer is concerned. In Pascal, these would be boolean, integer, real, character, subrange, and enumerated.
- Constructed types are arrays, records, and sets. In addition, pointers and functions can also be treated as constructed types.

7.2.1 Type Expressions

- The type of a language construct will be denoted by a type expression.
- Informally, a type expression is either a basic type or is formed by applying an operator called a type constructor to other type expressions.
- Type expressions can be defined as follows:
  1. A basic type is a type expression. A special basic type, type_error, will signal an error during type checking. Finally, a basic type void denoting the absence of a value allows statements to be checked.
  2. Since type expressions may be named, a type name is a type expression.
  3. A type constructor applied to type expressions is a type expression. Constructors include:
     (a) Arrays. If $T$ is a type expression, then $array(I, T)$ is a type expression denoting the type of an array with elements of type $T$ and index set $I$.
     (b) Products. If $T_1$ and $T_2$ are type expressions, then their Cartesian product $T_1 \times T_2$ is a type expression.
     (c) Records. The type of a record is in a sense the product of the types of its fields. The difference between a record and a product is that the fields of a record have names. Type checking of records can be done using the type expression formed by applying the constructor record to a tuple formed from field names and their associated types.
     (d) Pointers. If $T$ is a type expression, then $pointer(T)$ is a type expression denoting the type pointer to an object of type $T$.
     (e) Functions. Functions take values in some domain and map them into value in some range. This is denoted $domain \, values \rightarrow range \, values$.
  4. Type expressions may contain variables whose values are type expressions.
\begin{align*}
P & \rightarrow D ; E \\
D & \rightarrow D ; D | \text{id} : T \\
T & \rightarrow \text{char} | \text{integer} | \text{array} [\text{num}] \text{ of } T | \uparrow T \\
E & \rightarrow \text{literal} | \text{num} | \text{id} | E \mod E | E [E] | E \uparrow
\end{align*}

Figure 7.1: Grammar for source language

7.2.2 Type Systems

- A type system is a collection of rules for assigning type expressions to the various parts of a program.
- A type checker implements a type system.

7.2.3 Static and Dynamic Checking of Types

- Checking done by the compiler is static, while if it done at run-time, it is dynamic.
- 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.
- A language is strongly typed if its compiler can guarantee that the programs it accepts will execute without type errors.

7.2.4 Error Recovery

- It is important for a type checker to do something reasonable when an error is discovered.
- At the very least, the compiler must report the nature and location of the error.
- It is desirable for the type checker to recover from errors, so it can check the rest of the input.

7.3 Specification of a Simple Type Checker

- Consider the language in which identifiers must be declared before they are used.

7.3.1 A Simple Language

- Consider the grammar shown in Figure 7.1.
- Take a minute and discuss what can be in this language. Note that the declarations have to come before the usage of the variable.
<table>
<thead>
<tr>
<th>Production</th>
<th>SemanticRules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \rightarrow D; E$</td>
<td></td>
</tr>
<tr>
<td>$D \rightarrow D; D$</td>
<td></td>
</tr>
<tr>
<td>$D \rightarrow \text{id : } T$</td>
<td>$\text{addtype(id.entry, } T.\text{type})$</td>
</tr>
<tr>
<td>$T \rightarrow \text{char}$</td>
<td>$T.\text{type} = \text{char}$</td>
</tr>
<tr>
<td>$T \rightarrow \text{integer}$</td>
<td>$T.\text{type} = \text{integer}$</td>
</tr>
<tr>
<td>$T \rightarrow T_1$</td>
<td>$T.\text{type} = \text{pointer}(T_1.\text{type})$</td>
</tr>
<tr>
<td>$T \rightarrow \text{array [ num ] of } T_1$</td>
<td>$T.\text{type} = \text{array(num.val, } T_1.\text{type})$</td>
</tr>
</tbody>
</table>

Figure 7.2: Type checking scheme

<table>
<thead>
<tr>
<th>Production</th>
<th>SemanticRules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow \text{literal}$</td>
<td>$E.\text{type} = \text{char}$</td>
</tr>
<tr>
<td>$E \rightarrow \text{num}$</td>
<td>$E.\text{type} = \text{integer}$</td>
</tr>
<tr>
<td>$E \rightarrow \text{id}$</td>
<td>$E.\text{type} = \text{lookup(id.entry)}$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 \mod E_2$</td>
<td>$E.\text{type} = \text{if } E_1.\text{type} = \text{integer and } E_2.\text{type} = \text{integer}$</td>
</tr>
<tr>
<td>&amp; $\text{then integer else type_error}$</td>
<td></td>
</tr>
<tr>
<td>$E \rightarrow E_1[E_2]$</td>
<td>$E.\text{type} = \text{if } E_2.\text{type} = \text{integer and } E_1.\text{type} = \text{array(s, t)}$</td>
</tr>
<tr>
<td>&amp; $\text{then } t \text{ else type_error}$</td>
<td></td>
</tr>
<tr>
<td>$E \rightarrow E_1 \uparrow$</td>
<td>$E.\text{type} = \text{if } E_1.\text{type} = \text{pointer(t)}$</td>
</tr>
<tr>
<td>&amp; $\text{then } t \text{ else type_error}$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.3: Type system for expressions

- The basic types are: character and integer.
- The constructed types are: array and pointer.
- The attribute type is added to each symbol.
- The translation scheme is shown in Figure 7.2.
- Talk about the figure.

### 7.3.2 Type Checking of Expressions

- Consider Figure 7.3 which performs the type checking for expressions.
- Note that the synthesized attribute type for $E$ gives the type of the expression assigned by the type system for the expression generated by $E$.
- The function lookup returns the type of id.
7.3.3 Type Checking of Statements

- Statements do not have values, therefore a special type, void, can be assigned to them.
- If an error occurs within a statement, the type assigned to the statement is type_error.
- Consider the actions for statements shown in Figure 7.4.
- There is more checking that must occur on assignment statement. A check must be made to tell whether the left hand side can be assigned to. For example, you are not able to assign to a constant.

7.3.4 Type Checking Functions

- The application of functions can be seen in Figure 7.5.
- Notice that a function declaration would look like

  \[
  \text{root} : (\text{real} \rightarrow \text{real}) \times \text{real} \rightarrow \text{real}
  \]

  and would look like the following in Pascal.

  \[
  \text{function root (function f (real): real; x : real): real}
  \]


Algorithm 7.1 Structural Equivalence

function sequiv (Type : s, t) : boolean
begin
  if s and t are the same basic type
    return (TRUE);
  else if (s = array(s1, s2)) and (t = array(t1, t2))
    return (sequiv(s1, t1) and sequiv(s2, t2))
  else if (s = s1 × s2) and (t = t1 × t2)
    return (sequiv(s1, t1) and sequiv(s2, t2))
  else if (s = pointer(s1)) and (t = pointer(t1))
    return (sequiv(s1, t1))
  else if (s = s1 → s2) and (t = t1 → t2)
    return (sequiv(s1, t1) and sequiv(s2, t2))
  else
    return (FALSE);
end if
end

Figure 7.5: Algorithm for structural equivalence

7.4 Equivalence of Type Expressions

- We need to answer the question, "When are two type expressions equivalence?"

- The key issue is whether a name in a type expression stands for itself or whether it is an abbreviation for another type expression.

- We should remember that the notion of type equivalence needs to be implemented in the compiler in an efficient manner.

7.4.1 Structural Equivalence of Type Expressions

- Structural equivalence – two expressions are either the same basic type, or are formed by applying the same constructor to structurally equivalence types.

- Consider the algorithm given in Figure 7.6 that checks for structural equivalence.

7.4.2 Names for Type Expressions

- In some languages, types can be given names.

- Consider the following situation.

```plaintext
type link = ^cell;
var next : link;
```

Type Checking
last : link;
    p : cell;
    q, r : cell;

- Do the variables \texttt{next, last, p, q, r} all have identical types?
- The answer depends on the implementation.
- When names are allowed in type expressions, two notion of equivalence of type expressions arise, depending on the treatment of names:
  - \textit{Name equivalence} -- Each type name is viewed as a distinct type, so two type expressions are name equivalent if and only if they are identical.
  - \textit{Structural equivalence} -- Names are replaced by the type expressions they define, so two type expressions are structurally equivalent if they represent two structurally equivalent type expressions when all names have been substituted out.
- Consider the above example in light of the two definitions of equivalence given.

7.5 Type Conversion

- Since the representation of integers and reals is different within a computer, and different machine instructions are used for operations on integers and reals, the compiler may have to first convert one of the operands of $+$ to ensure that both operands are of the same type when the addition takes place.
- The language definition determines when conversion is necessary.

7.5.1 Coercions

- Conversion from one type to another is said to be \textit{implicit} if it is to be done automatically by the compiler.
- Implicit type conversions are also called \textit{coercions}.
- Conversion is said to be \textit{explicit} if the programmer must write something to cause the conversion.
- Consider the evaluation of expressions shown in Figure 7.7.
- It is possible to do dump things in the conversion. For example, it was found that

\[
\text{for } i := 1 \text{ to } N \text{ do } x[i] := 1
\]

took $48.8N$ microseconds, while the fragment
<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow \text{num}$</td>
<td>$E.type = \text{integer}$</td>
</tr>
<tr>
<td>$E \rightarrow \text{num} \cdot \text{num}$</td>
<td>$E.type = \text{real}$</td>
</tr>
<tr>
<td>$E \rightarrow \text{id}$</td>
<td>$E.type = \text{lookup(id.entry)}$</td>
</tr>
</tbody>
</table>
| $E \rightarrow E_1 \text{ op } E_2$ | $E.type = \text{if } E_1.type = \text{integer and } E_2.type = \text{integer}$ then integer  
|                  | else if $E_1.type = \text{integer and } E_2.type = \text{real}$ then real   |
|                  | else if $E_1.type = \text{real and } E_2.type = \text{integer}$ then real   |
|                  | else if $E_1.type = \text{real and } E_2.type = \text{real}$ then real       |
|                  | else type_error                                                               |

Figure 7.7: Coercion of expressions

```plaintext
for i := 1 to N do x[i] := 1.0
```

took 5.4N microseconds. Why?

### 7.6 Overloading of Functions and Operators

- An *overloaded* symbol is one that has different meanings depending on its context.
- Overloading is *resolved* when a unique meaning for an occurrence of an overloaded symbol is determined.
- The resolution of overloading is sometimes referred to as *operator identification* because it determines which operation an operator symbol denotes.

#### 7.6.1 Set of Possible Types for a Subexpression

- It is not always possible to resolve overloading by looking only at the arguments of a function.
- Consider the following possible Ada declaration.

```plaintext
function "*" (i, j : integer) return complex;
function "*" (i, j : complex) return complex;
```

We then have the follow possible types for *.
<table>
<thead>
<tr>
<th>Production</th>
<th>SemanticRules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E' \rightarrow E$</td>
<td>$E'.types = E.types$</td>
</tr>
<tr>
<td>$E \rightarrow id$</td>
<td>$E.types = \text{lookup(id.entry)}$</td>
</tr>
<tr>
<td>$E \rightarrow E_1(E_2)$</td>
<td>$E.types = { t \mid \text{there exists an } s \text{ in } E_2.types \text{ such that } s \rightarrow t \text{ is in } E_1.types }$</td>
</tr>
</tbody>
</table>

Figure 7.8: Type checking and overloading of operators

\[
\begin{align*}
\text{integer} \times \text{integer} & \rightarrow \text{integer} \\
\text{integer} \times \text{integer} & \rightarrow \text{complex} \\
\text{complex} \times \text{complex} & \rightarrow \text{complex}
\end{align*}
\]

- Given the expression $2 \times 3$ it is impossible to know which of the above it correct. If there is more context in the expression, for example the preceding expression is multiplied by another integer or another complex number, then we could choose.

- If we generalize the rules we had for type checking function previously, we come up with what is shown in Figure 7.8.

### 7.7 Narrowing the Set of Possible Types

- Ada requires a complete expressions to have a unique type. Given a unique type from the context, we can narrow down the type choices for each subexpression. If this process does not result in a unique type for each subexpression, then a type error is declared for the expression.

- Consider the syntax-directed translation shown in Figure 7.9. Note that the attribute type is synthesized while unique is inherited.
<table>
<thead>
<tr>
<th>Production</th>
<th>SemanticRules</th>
</tr>
</thead>
</table>
| $E' \rightarrow E$ | $E'.types = E.types$
| | $E.unique = \text{if } E'.types = \{t\} \text{ then } t \text{ else } \text{type.error}$
| | $E'.code = E.code$
| $E \rightarrow id$ | $E.types = \text{lookup(id.entry)}$
| | $E.code = \text{gen(id.lexeme} ' ' E.unique$
| $E \rightarrow E_1(E_2)$ | $E.types = \{t \mid \text{there exists an } s \text{ in } E_2.types$
| | $\quad \quad \quad \quad \quad \quad \text{such that } s \rightarrow t \text{ is in } E_1.types \}$
| | $t = E.unique$
| | $S = \{s \mid s \in E_2.types \text{ and } s \rightarrow t \in E_1.types\}$
| | $E_2.unique = \text{if } S = \{s\} \text{ then } s \text{ else } \text{type.error}$
| | $E_1.unique = \text{if } S = \{s\} \text{ then } s \rightarrow t \text{ else } \text{type.error}$
| | $E.code = E_1.code || E_2.code || \text{gen('apply} ' ' E.unique$}

Figure 7.9: Type checking