Static Checking

Static Checking refers to the compile-time checking of programs to ensure that the syntactic and semantic conventions of the source language are being followed. Examples include:

- **Type Checks**: operators and operands must have "compatible" types.

- **Flow-of-control Checks**: statements that cause flow of control to leave a construct must have some place where control can be transferred, e.g. break statements in C.

- **Uniqueness Checks**: A language may dictate that in some contexts, an entity can be defined exactly once. E.g.: identifiers in declarations, case statement labels, etc.

- **Name-related Checks**: Sometimes, the same name must appear two or more times, e.g. in Ada a loop or block can have a name that must appear at the beginning and at the end.

Some or all of these activities can be folded into other activities, e.g. some checking can be integrated with parsing.
6.4. Data Types and Type Checking

- A *data type* is a set of values together with a set of operations that can be performed on them.

- The purpose of *type checking* is to verify that any operations performed on a value are, in fact, permissible.

- The type of an identifier is typically available from declarations, but we may have to keep track of the type of intermediate values.

- A language usually provides a set of *base types* that it supports, together with ways to construct other types using *type constructors*.

- We have to be able to represent types that may be defined in a program so that we can reason about them.
  
  This is usually done using *type expressions*.

6.4.1. Type Expressions and Type Constructors

A *type expression* denotes the type of a language construct. E.g.:

1. A base type is a type expression, e.g.: boolean, char, int, float.

2. A type name (e.g., the name of a record type) is a type expression.
3. A type constructor applied to type expressions is a type expression. E.g.:
   (a) arrays : if \( T \) is a type expression and \( I \) is a range of integers then \( array(I,T) \) is a type expression.
   (b) records : if \( T_1, \ldots, T_n \) are type expressions and \( f_1, \ldots, f_n \) are field names, then
       \( record((f_1,T_1), \ldots, (f_n,T_n)) \)
       is a type expression.
   (c) pointers : if \( T \) is a type expression then \( pointer(T) \)
       is a type expression.
   (d) functions : if \( T_1, \ldots, T_n, T \) are type expressions then so is
       \( (T_1, \ldots, T_n) \rightarrow T \).

**Notions of Type Equivalence**

1. **Name Equivalence**: In some languages, e.g., Pascal, types can be given names. Name Equivalence views each distinct name as a distinct type. So two type expressions are name equivalent if and only if they are identical.

2. **Structural Equivalence**: Two expressions are structurally equivalent if they have the same structure, i.e., if they are formed by applying the same constructor to structurally equivalent type expressions.

**Example**: In the Pascal fragment

\[
\text{type nextptr} = \uparrow \text{node};
\text{prevptr} = \uparrow \text{node};
\text{var} \ p : \text{nextptr};
\text{q} : \text{prevptr};
\]

\( p \) is not name-equivalent to \( q \), but \( p \) and \( q \) are structurally equivalent.
**Compile-Time Representation of Types**

**Issue**: Need to represent type expressions in a way that is both easy to construct and easy to check.

**Type Graphs**: Represent the type expression as a graph. E.g.:

- Basic types can have predefined "internal values", e.g. small integer values.
- Named types can be represented using a pointer into a hash table.
- Composite type expressions: the node for $f(T_1, \ldots, T_n)$ contains a value representing the type constructor $f$, and pointers to the nodes for the expressions $T_1, \ldots, T_n$.

Recursive types, e.g. linked lists and trees, yield type graphs with cycles.

---

**Example**: var x, y : array[10 .. 20] of integer;

![Type Graph Example Diagram]
Type Checking of Expressions

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
</table>
| $E \rightarrow \text{id}$ | if ( declared(\text{id}.name) )  
then  
$E\.type := \text{lookup(\text{id}.name)}\.type$  
else  
$E\.type := \text{error}()$; |
| $E \rightarrow \text{intcon}$ | $E\.type := \text{integer}$, |
| $E \rightarrow E_1 + E_2$ | if ($E_1\.type == \text{integer}$ AND  
$E_2\.type == \text{integer}$)  
then  
$E\.type := \text{integer}$;  
else  
$E\.type := \text{error}()$; |

May have automatic coercion, e.g.:

<table>
<thead>
<tr>
<th>$E_1.type$</th>
<th>$E_2.type$</th>
<th>$E.type$</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>integer</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>float</td>
<td>integer</td>
<td>float</td>
</tr>
<tr>
<td>float</td>
<td>float</td>
<td>float</td>
</tr>
</tbody>
</table>
Type-Checking Statements

Assignment: Semantic Rules:

\[ S \rightarrow Lval := Rval \]
\[ \{ \text{check_types}(Lval.type, Rval.type) \} \]

Note that \( Lval \) can be a variable, or it may be a more complicated expression, e.g., a dereferenced pointer, an array element, a record field, etc.

Type checking involves ensuring that

- \( Lval \) is of a type that can be assigned to, e.g., procedures cannot be assigned to (except for language-specific exception, e.g., in Pascal assigning to the identifier \( foo \) while in the body of a function \( foo() \)).
- The types of \( Lval \) and \( Rval \) are “compatible”, i.e., that the language provides rules for coercion of the type of \( Rval \) to the type of \( Lval \).

Loops, Conditionals: Semantic Rules:

\[ \text{Loop} \rightarrow \text{while } E \text{ do } S \]
\[ \{ \text{check_types}(E.type, \text{bool}) \} \]
\[ \text{Cond} \rightarrow \text{if } E \text{ then } S \text{ else } S \]
\[ \{ \text{check_types}(E.type, \text{bool}) \} \]