Type Checking

- Type checker verifies that the type of a construct (constant, variable, array, list, object) matches what is expected in its usage context.
- E.g., Java's `%` (modulo) operator expects two integers, so $3 \% 4.5$ is a type error.
- Some operators (+, -, *, /) are "overloaded": i.e., they can apply to objects of different types (int, real).
- Functions may be polymorphic; i.e., accept arguments of different types.

6.1 Type Systems

- Type checker needs to know about
  - Syntactic constructs in language (e.g., operators)
  - Basic types of language (int, real)
  - Rules for assigning types to constructs
- E.g., "if both operands of + are int, result is int"
- Leads to recursively defined type expression:
  - Basic type: int, real
  - Type variable (α, β, ...)
  - Result of applying type constructor to type expression

Type Constructors

- Cartesian Product: If $T_1$ and $T_2$ are type expressions, then $T_1 \times T_2$ is a type expression (e.g., ML tuples)
- Function: If $T_1$ and $T_2$ are type expressions, then $T_1 \rightarrow T_2$ is a type expression (e.g., ML functions)
Static vs. Dynamic Type Checking

- **Static**: Done at compile time (e.g., Java)
- **Dynamic**: Done at run time (e.g., Scheme)
- **Sound** type system is one where any program that passes the static type checker cannot contain run-time type errors. Such languages are said to be *strongly typed*.
- Some errors can only be detected at run-time; e.g., array out-of-bounds:
  ```java
  int [] a = new int [10];
  for (int i=0; i<20; ++i)
  a[i] = i;
  ```

6.2 Specification of a Simple Type Checker

- Consider a language (e.g., Java) that requires that an identifier be declared with a type before the variable is used.
- For simplicity, we'll declare all identifiers before using them in a single expression:
  ```
  P → D ; E
  D → D ; D | T id
  T → char | int | T [ num ]
  E → id | E mod E | E [ E ]
  ```

Type Checking of Statements

- A more realistic language would also have statements (whose type is trivially `void`):
  ```
  P → D ; S
  S → id := E ( S.type := if id.type = E.type then void else type_error )
  S → if E then S₁ ( S.type := if E.type = boolean then S₁.type else type_error )
  S → S₁ ; S₁ ( S.type := if S₁.type = void and S₂.type = void then void else type_error )
  ```

- Add syntax-directed definitions to type-check expressions:
  ```
  D → T id ( addtype ( id.entry, T.type ) )
  T → char ( T.type := char )
  T → int ( T.type := integer )
  T → T₁ [ num ] ( T.type := array (0..num.val-1, T₁.type ) )
  E → id ( E.type := lookup ( id.entry ) )
  E → E mod E₁ ( E.type := if E₁.type = integer and E.type = integer then integer else type_error )
  E → E₁ [ E₂ ] ( E.type := if E₁.type = integer and E₁.type = array(s,t) then t else type_error )
  ```

- Where `addtype`, `lookup` are symbol-table methods, `array` is a type constructor
Type Checking of Functions

- We can declare function types using a rule:
  \[ T \rightarrow T_1 \rightarrow \cdots \rightarrow T_j \rightarrow \text{function} \]
  where \( \text{function} \) is another type constructor.
- ML supports this, but in practice function types are usually inferred.
- In either case, we need to type-check function application:
  \[ E \rightarrow E_1 (E_j) \{ E \text{type} := \{ \begin{array}{ll}
    \text{if } E_j \text{type} = s \text{ and } \\
    E \text{type} = \text{function}(s,t) \text{ then } t \\
    \text{else type_error}
  \end{array} \} \]

6.3 Equivalence of Type Expressions

- Need a way of implementing checks like \( E_i \text{type} = s \)
- Structural Equivalence: Recursive definition:
  \[
  \begin{align*}
  \text{equiv}(s,t)
  & \text{ : boolean} \\
  & \text{if } s \text{ and } t \text{ are the same basic type then return true} \\
  & \text{else if } s = \text{array}(s_1,s_2) \text{ and } t = \text{array}(t_1, t_2) \text{ then} \\
  & \quad \text{return equiv}(s_1, t_1) \text{ and equiv}(s_2, t_2) \\
  & \text{else if } s = \text{cartesian}(s_1,s_2) \text{ and } t = \text{cartesian}(t_1, t_2) \text{ then} \\
  & \quad \text{return equiv}(s_1, t_1) \text{ and equiv}(s_2, t_2) \\
  & \text{else if } s = \text{function}(s_1, s_2) \text{ and } t = \text{function}(t_1, t_2) \text{ then} \\
  & \quad \text{return equiv}(s_1, t_1) \text{ and equiv}(s_2, t_2) \\
  & \text{else return false}
  \end{align*}
\]

6.5 Overloading of Functions and Operators

- Some operators (+, - , *, /) are "overloaded"; i.e., they can apply to objects of different types (int, real).
- Overloading is resolved when we determine the actual type.
- Arguments alone do not always resolve operator's type; e.g., in Java type of \( 3*5 \) may resolve to either int or double, depending on context: \( 3*5+2 \) (int) or \( 3*5+2.0 \) (double)
- We allow the .type attribute to be a set of types, and add code to narrow this set to a single type for a given expression.
- Add an attribute .unique to contain unique type, and a special symbol \( E \) to support narrowing:

\[
E \rightarrow E \{ E \text{types} := E \text{types} \\
& \quad E \text{unique} := \text{if } E \text{types} = \{t\} \text{ then else type_error} \}
\]

\[
E \rightarrow \text{id} \{ E \text{types} := \{ \text{lookup (id.entry) } \} \}
\]

\[
E \rightarrow E_j (E_j) \{ E \text{types} := s \mid 3 \in E_j \text{types such that} \\
& \quad \text{function}(s, t) \in E \text{types} \\
& \quad t := E \text{unique} \\
& \quad S := \{ s \mid s \in E_j \text{types and function}(s,t) \in E \text{types} \} \\
& \quad E_j \text{unique} := \text{if } S = \{t\} \text{ then else type_error} \\
& \quad E_j \text{unique} := \text{if } S = \{s\} \text{ then function}(s,t) \\
& \quad \text{else type_error} \}
\]