NPTEL ONLINE CERTIFICATION COURSES

Compiler Design

Type Checking

Santanu Chattopadhyay
Electronics and Electrical Communication Engineering

CONCEPTS COVERED

- What is a Type Checking
- Static vs. Dynamic Checking
- Type Expressions
- Type Equivalence
- Type Conversion
- Phases of a Compiler
- Conclusion

What is Type Checking

- One of the most important semantic aspects of compilation
- Allows the programmer to limit what types may be used in certain circumstances
- Assigns types to values
- Determines whether these values are used in an appropriate manner
- Simplest situation: check types of objects and report a type-error in case of a violation
- More complex: incorrect types may be corrected (type coercing)

Static vs. Dynamic Checking

- Static Checking
  - Type checking done at compile time
  - Properties can be verified before program run
  - Can catch many common errors
  - Desirable when faster execution is important

- Dynamic Checking
  - Performed during program execution
  - Permits programmers to be less concerned with types
  - Mandatory in some situations, such as, array bounds check
  - More robust and clearer code

Type Expressions

- Used to represent types of language constructs
- A type expression can be
  - Basic type: integer, real, char, Boolean and other atomic types that do not have internal structure. A special type, type-error is used to indicate type violations
  - Type name
  - Type constructor applied to a list of type expressions

Type Expressions

- Arrays are specified as array(I,T), where T is a type and I is an integer or a range of integers. For example, C declaration “int a[100]” identifies type of a to be array(100, integer)

- If T1 and T2 are type expressions, T1 × T2 represents “anonymous records”. For example, an argument list passed to a function with first argument integer and second real, has type integer × real

Type Expressions

- Named records are products with named elements. For a record structure with two named fields – length (an integer) and word (of type array(10, char)), the record is of type

  record((length × integer) × (word × array(10, character)))
- If T is a type expression, pointer(T) is also a type expression, representing objects that are pointers to objects of type T
- Function maps a collection of types to another, represented by D → R, where D is the domain and R is the range of the function.

Type Expressions

- Type expression “integer × integer → character” represents a function that takes two integers as arguments and returns a character value
- Type expression “integer → (real →character)” represents a function that takes an integer as an argument and returns another function which maps a real number to a character

Type Systems

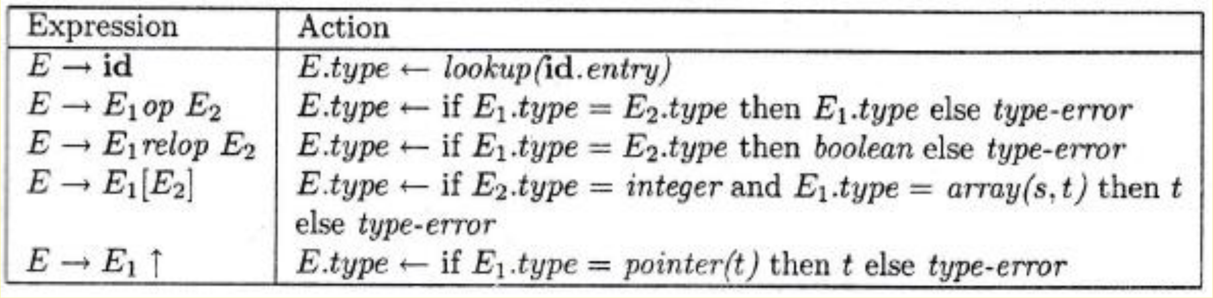
- Type system of a language is a collection of rules depicting the type expression assignments to program objects
- Usually done with syntax directed definition
- ‘Type checker’ is an implementation of a type system

Strongly Typed Language

- Compiler can verify that the program will execute without any type errors
- All checks are made statically
- Also called a sound type system
- Completely eliminates necessity of dynamic type checking
- Most programming languages are weakly typed
- Strongly typed languages put lot of restrictions
- There are cases in which a type error can be caught dynamically only
- Many languages also allow the user ti override the system

Type Checking of Expressions

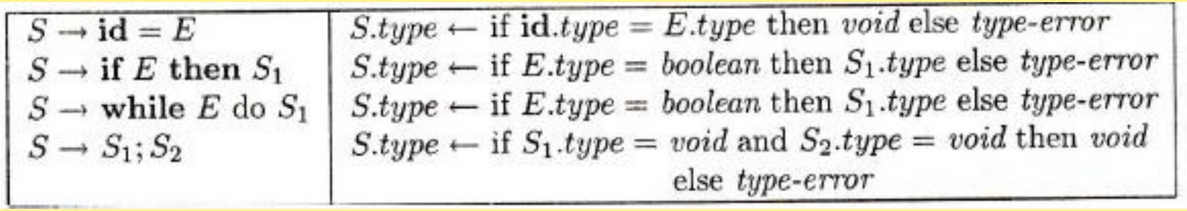
- Use synthesized attribute ‘type’ for the nonterminal E representing an expression



Expression	Action
$E \rightarrow \textbf{id}$	$E.type \leftarrow lookup(\textbf{id}.entry)$
$E \rightarrow E_1\, op\, E_2$	$E.type \leftarrow$ if $E_1.type = E_2.type$ then $E_1.type$ else *type-error*
$E \rightarrow E_1\, relop\, E_2$	$E.type \leftarrow$ if $E_1.type = E_2.type$ then *boolean* else *type-error*
$E \rightarrow E_1[E_2]$	$E.type \leftarrow$ if $E_2.type = integer$ and $E_1.type = array(s,t)$ then t else *type-error*
$E \rightarrow E_1 \uparrow$	$E.type \leftarrow$ if $E_1.type = pointer(t)$ then t else *type-error*

Type Checking of Statements

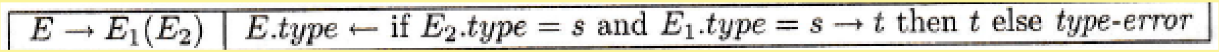
- Statements normally do not have any value, hence of type void
- For propagating type error occurring in some statement nested deep inside a block, a set of rules needed



$S \rightarrow \mathbf{id} = E$	$S.type \leftarrow$ if $\mathbf{id}.type = E.type$ then *void* else *type-error*
$S \rightarrow \mathbf{if}\ E\ \mathbf{then}\ S_1$	$S.type \leftarrow$ if $E.type = boolean$ then $S_1.type$ else *type-error*
$S \rightarrow \mathbf{while}\ E\ \mathrm{do}\ S_1$	$S.type \leftarrow$ if $E.type = boolean$ then $S_1.type$ else *type-error*
$S \rightarrow S_1; S_2$	$S.type \leftarrow$ if $S_1.type = void$ and $S_2.type = void$ then *void* else *type-error*

Type Checking of Functions

- A function call is equivalent to the application of one expression to another



$E \rightarrow E_1(E_2)$	$E.type \leftarrow$ if $E_2.type = s$ and $E_1.type = s \rightarrow t$ then t else $type\text{-}error$

Type Equivalence

- It is often needed to check whether two type expressions ‘s’ and ‘t’ are same or not
- Can be answered by deciding equivalence between the two types
- Two categories of equivalence
  - Name equivalence
  - Structural equivalence