Semantic Analysis

Type Checking

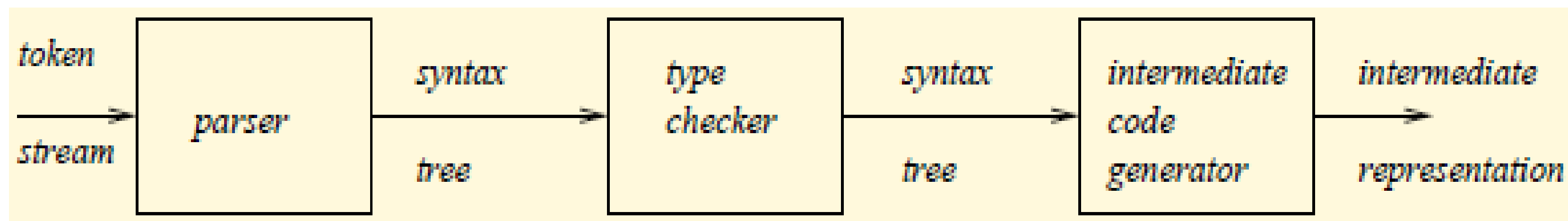
Prepared By:
Prashant Gautam
2021

Type Checking

- A type is a set of values together with a set of operations that can be performed on them
- ***Type checking*** is checking that each operation in a program receives appropriate number of arguments of appropriate types in appropriate order.
- The purpose of type checking is to verify that operations performed on a value are in fact permissible.
- Certain operations are legal for values of each type
  - It doesn't make sense to add a function pointer and an integer in C.
  - It does make sense to add two integers.
- The type of an identifier is typically available from declarations, but we may have to keep track of the type of intermediate expressions.
- **Type errors** arise when operations are performed on values that do not support that operation.

Type Systems

- A language's type system specifies which operations are valid for which types.
- Type systems provide a concise formalization of the semantic checking rules.
- A type system defines a set of types and rules to assign types to programming language constructs like informal type system rules, for example "if both operands of addition are of type integer, then the result is of type integer".
- *Type Checking* is the process of checking that the program obeys the type system.
- A type checker implements type system.
- A sound type system eliminates run-time type checking for type errors.
  - Memory errors: Reading from an invalid pointer, etc.
  - Violation of abstraction boundaries.



Type Checking Overview

Three kinds of languages:

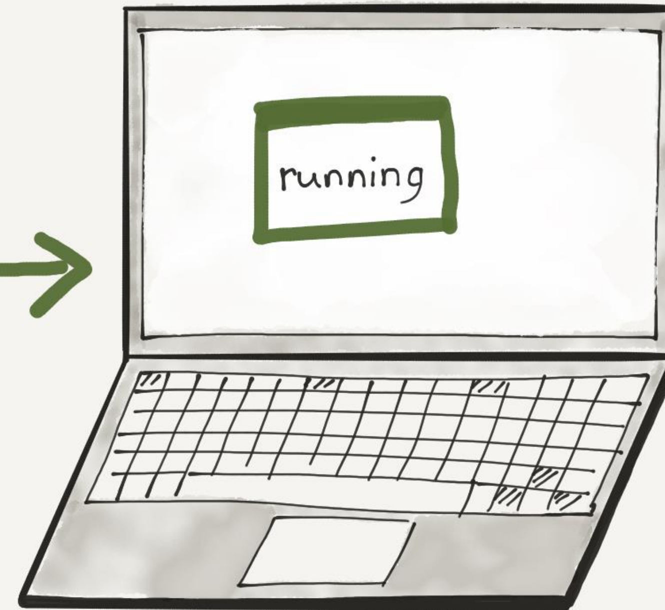
- **Statically typed**: All or almost all checking of types is done as part of compilation (C, ML, Java)
- **Dynamically typed**: Almost all checking of types is done as part of program execution (Scheme, Prolog)
- **Untyped**: No type checking (machine code)

- **Static typing proponents say:**
  - Static checking catches many programming errors at compile time
  - Avoids overhead of runtime type checks
- **Dynamic typing proponents say:**
  - Static type systems are restrictive
  - Rapid prototyping easier in a dynamic type system

Compile time

Check types here
(Static typing)

Runtime



Check types here
(Dynamic typing)

Static Checking

- Refers to the compile-time checking of programs in order to ensure that the semantic conditions of the language are being followed
- Examples of static checks include:
  - Type checks
  - Flow-of-control checks
  - Uniqueness checks
  - Name-related checks
- 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., identifier declarations, labels, values in case expressions
- Name-related checks: Sometimes the same name must appear two or more times; e.g., a loop or block can have a name that must then appear both at the beginning and at the end

Flow-of-control checks

```
myfunc()
{ …
  break; // ERROR
}
```

```
myfunc()
{ …
  while (n)
  { …
    if (i>10)
      break; // OK
  }
}
```

```
myfunc()
{ …
  switch (a)
  { case 0:
      …
      break; // OK
    case 1:
      …
  }
}
```

Uniqueness checks

```
myfunc()
{ int i, j, i; // ERROR
  ...
}
```

```
cnufym(int a, int a) // ERROR
{    ...
}
```

```
struct myrec
{ int name;
};
struct myrec // ERROR
{ int id;
};
```

Name-related checks

```
LoopA: for (int I = 0; I < n; I++)
       { …
         if (a[I] == 0)
           break LoopB;
         …
       }
```

Type Expression

- A language usually provides a set of base types that it supports together with ways to construct other types using type constructors
- Through type expressions we are able to represent types that are defined in a program
- A base type is a type expression
  - a primitive data type such as integer, real, char, boolean, …
  - type-error signal an error during type checking
  - void : no type
- A type name (e.g., a record name) is a type expression
- A type constructor applied to type expressions is a type expression. E.g.,
  - arrays: If T is a type expression and I is a range of integers, then array(I,T) is a type expression
  - records: If T1, …, Tn are type expressions and f1, …, fn are field names, then record((f1,T1),…,(fn,Tn)) is a type expression
  - pointers: If T is a type expression, then pointer(T) is a type expression Ex: pointer(int)
  - functions: If T1, …, Tn, and T are type expressions, then so is (T1,…,Tn) →T. Ex: int→int represents the type of a function which takes an int value as parameter, and return type is also int.

A Simple Type Checking System

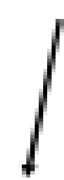
$P \rightarrow D \,;\, S$

$D \rightarrow D \,;\, D \mid \textbf{id} : T$

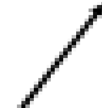
$T \rightarrow \textbf{boolean} \mid \textbf{char} \mid \textbf{integer} \mid \textbf{array [num] of } T \mid \uparrow T$

$S \rightarrow \textbf{id} = E \mid \textbf{if } E \textbf{ then } S \mid \textbf{while } E \textbf{ do } S \mid S \,;\, S$

$E \rightarrow \textbf{true} \mid \textbf{false} \mid \textbf{literal} \mid \textbf{num} \mid \textbf{id} \mid E \textbf{ and } E \mid E + E \mid E\,[\,E\,] \mid E \uparrow$



Pointer to T



Pascal-like pointer dereference operator

This grammar generate the language with declarations followed by statements like

key : integer; key = 199

.

if true then key = 100;

Type checking for expression

- 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.
- The following figure shows the type checking for the expressions.

Production	*SemanticRules*
$E \rightarrow \textbf{literal}$	$E.type$ = *char*
$E \rightarrow \textbf{num}$	$E.type$ = *integer*
$E \rightarrow \textbf{id}$	$E.type$ = *lookup* (**id**.*entry*)
$E \rightarrow E_1$ **mod** E_2	$E.type$ = **if** $E_1.type$ = *integer* **and** $E_2.type$ = *integer* **then** *integer* **else** *type_error*
$E \rightarrow E_1[E_2]$	$E.type$ = **if** $E_2.type$ = *integer* **and** $E_1.type$ = *array(s, t)* **then** t **else** *type_error*
$E \rightarrow E_1 \uparrow$	$E.type$ = **if** $E_1.type$ = *pointer(t)* **then** t **else** *type_error*

Type checking for statements

- In some languages statements have a type associated with them, while some other languages don't assign types to statements.
- In the latter case, statements are given a type void to distinguish a type safe statement with one which has a type error.
- if an error occurs within a statement, then the type assigned to this statement is type_error.

Production	Semantic Rules
$S \rightarrow \textbf{id} = E$	$S.type = \textbf{if } \textbf{id}.type = E.type$ $\textbf{then } void \textbf{ else } type_error$
$S \rightarrow \textbf{if } E \textbf{ then } S_1$	$S.type = \textbf{if } E.type = boolean$ $\textbf{then } void \textbf{ else } type_error$
$S \rightarrow \textbf{while } E \textbf{ do } S_1$	$S.type = \textbf{if } E.type = boolean$ $\textbf{then } void \textbf{ else } type_error$
$S \rightarrow S_1 ; S_2$	$S.type = \textbf{if } S_1.type = void \textbf{ and } S_2.type = void$ $\textbf{then } void \textbf{ else } type_error$

Type checking for functions

- A function to an argument can be captured by production:

  T→T->T — *Function type declaration*

  E→E(E) — *Function call*

Production	*SemanticRules*
$T \rightarrow T_1$ " $\rightarrow$ " T_2	$T.type = T_1.type \rightarrow T_2.type$
$E \rightarrow E_1(E_2)$	$E.type =$ **if** $E_2.type = s$ **and** $E_1.type = s \rightarrow t$ **then** t **else** $type_error$

Example:

```
v : integer;
odd : integer -> boolean;
if odd(3) then
    v := 1;
```

Type Conversion and Coercion

- Since representation of integer and real is different within a computer, the different machine instructions are used for operations on integers and reals. Often if different parts of an expression are of different types then type conversion is required.
- For example, in the expression: z = x + y what is the type of z if x is integer and y is real ?
- Compilers have to convert one of the them to ensure that both operand of same type!
- In many language Type conversion is explicit, for example using type casts i.e. must be specify as inttoreal(x)
- Type conversions which happen implicitly is called coercion. Implicit type conversions are carried out by the compiler recognizing a type incompatibility and running a type conversion routine (for example, something like inttoreal(int)) that takes a value of the original type and returns a value of the required type.
- The coercion of expressions is given in following figure.

Type Conversion and Coercion (Contd.)

Production	Semantic Rules
$E \rightarrow$ **num**	$E.type$ = *integer*
$E \rightarrow$ **num . num**	$E.type$ = *real*
$E \rightarrow$ **id**	$E.type$ = *lookup*(**id**.*entry*)
$E \rightarrow E_1$ **op** E_2	$E.type$ = **if** $E_1.type$ = *integer* **and** $E_2.type$ = *integer* **then** *integer* **else if** $E_1.type$ = *integer* **and** $E_2.type$ = *real* **then** *real* **else if** $E_1.type$ = *real* **and** $E_2.type$ = *integer* **then** *real* **else if** $E_1.type$ = *real* **and** $E_2.type$ = *real* **then** *real* **else** *type_error*

Approaches for Type Equivalence

1. Name Equivalence
2. Structure Equivalence

Problem: When in E1.type = E2.type?

- We need a precise definition for type equivalence
- Interaction between type equivalence and type representation

Example:

```
type vector = array [1..10] of real
type weight = array [1..10] of real
var x, y: vector; z: weight
```

Name Equivalence: When they have the same name.

- x, y have the same type; z has a different type.

Structural Equivalence: When they have the same structure.

- x, y, z have the same type.

More Precisely

- **Name equivalence:** In many 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.
- In the Pascal fragment

```
type nextptr = ^node;
prevptr = ^node;
var p : nextptr;
q : prevptr;
```

- p is not name equivalent to q

- Structural equivalence: Two expressions are structurally equivalent if and only if they have the same structure; i.e., if they are formed by applying the same constructor to structurally equivalent type expressions.
- In the Pascal fragment

```
type nextptr = ^node;
prevptr = ^node;
var p : nextptr;
q : prevptr;
```

- p is not name equivalent to q,
- but p and q are structurally equivalent.

Old Question

6. Differentiate between static and dynamic type checking. How can we carry out type checking for the following expression using syntax directed definition?

   S->id=E

   S->if E then S_1

   S->while E do S_1

   S->S_1;S_2

7. What is type checking? Explain about the static and dynamic type checking.
8. What is type equivalence? Explain name equivalence and structural equivalence with eg.
9. What is type system? Give an example.
10. Write Syntax Directed Definitions to carry out the Type Checking for the following Expression.

    E→ id | E1 op E2 | E1 relop E2 | E [E2] | E pointer.

Most Important

Table 4.1: Type checking expressions

Expression	Action
$E \rightarrow \mathbf{id}$	$E.type \leftarrow lookup(\mathbf{id}.entry)$
$E \rightarrow E_1\, op\ E_2$	$E.type \leftarrow$ 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* t 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*

Table 4.2: Type checking statements

$S \rightarrow \mathbf{id} = E$	$S.type \leftarrow$ if $\mathbf{id}.type = E.type$ then *void* else *type-error*
$S \rightarrow$ **if** E **then** S_1	$S.type \leftarrow$ if $E.type = boolean$ then $S_1.type$ t else *type-error*
$S \rightarrow$ **while** E 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*

Table 4.3: Type checking functions

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