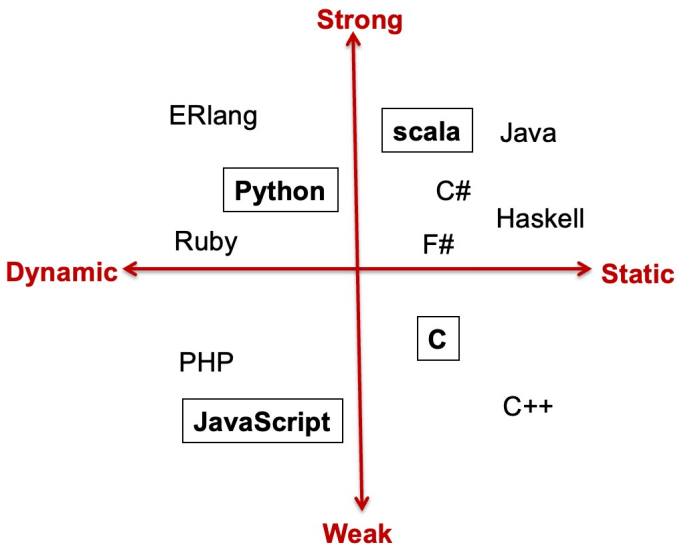
CMPSC 160
Translation of Programming Languages

Lecture 10: Type System

Recap on Type Analysis

- Type: Definition and Benefits.
- Two notions of typing for programming languages are to be distinguished:
  - static vs. dynamic
  - strong vs. weak



- **Static** (compilation time) Type Checking
  - Type equivalence
  - Type Inference for expressions
  - **Ad-hoc Syntax-directed Translation on a simple language**

Our focus: Static Type Checking

- **Key difficulty:** We need to do **type inference** and it could be difficult for the following two issues.
  - For **expressions**

    The following example:

    int f(int x) { return x+1; };

    int g(int y) { return f(y+1)*2; };

    We need to do type inference for y+1, and f(y+1)*2. The expression could be complex and involve implicit conversions.
  - For **user-defined types**

    Example on next page

User-defined types

```
typedef struct {
        int data[100];
        int count;
        } Stack;

typedef struct {
        int data[100];
        int count;
        } Set;

Stack x, y;
Set r, s;
```

```
x = y;
r = s;
```

Always correct!!!

```
x = r;
```

This depends on how we define equivalence!

Suppose this is c language, can we have x = r???

NO.

Type Equivalence

- **Name Equivalence**: Types are identical only if names match Treat named types as basic types.
- **Structural equivalence:** Replace the named types by their definitions and recursively check the substituted trees. That is, the two types have the same definitional structure.

```
typedef struct {
        int data[100];
        int count;
        } Stack;

typedef struct {
        int data[100];
        int count;
        } Set;

Stack x, y;
Set r, s;
```

```
x = y;
r = s;
```

Always correct!!!

```
x = r;
```

This depends!

Yes for structural equivalence

No for name equivalence

Language Examples: Name and Structure Equivalence

- Which notion of Equivalence do Programming Languages Use?
  - C and C++: Both. For structs and classes, it uses name equivalence. For pointers and arrays, it use structural equivalence. For arrays, size is ignored. use structural equivalence.
  - Java uses name equivalence. In particular for classes, it uses name equivalence except that a subtype may be used where a parent type is expected.
  - Modula-3 uses structural equivalence.
  - …

Type System

- **Type system:** the set of types in a programming language, along with the rules that use types to specify program behavior, are collectively called a type system.
  - **A set of base types, or built-in types**
    - Programming languages typically include base types for numbers (int, float), characters, Booleans, etc.
  - **Rules for constructing new types from the existing types**
    - array, function, pointer, product
    - Programmers need higher-level abstractions to combine and aggregate objects and to derive types for the resulting objects.
  - **Rules for type inference:** the process of determining a type for each name and each expression in the code.
  - **Rules for type equivalence check:** name equivalence and structural equivalence.

Type Checking for a Tiny Language

- Given the following grammar for a very simple language

$P \rightarrow D \ ; \ S$

$D \rightarrow D \ ; \ D \mid \text{id} : T$

$T \rightarrow \text{char} \mid \text{int} \mid \text{array [num] of } T \mid \text{pointer } T \mid \text{function } T \text{ to } T$

$E \rightarrow \text{literal} \mid \text{num} \mid \text{id} \mid E \text{ mod } E \mid E [E] \mid * E \mid E (E) \mid E = E$

$S \rightarrow \text{id} := E \mid \text{if } E \text{ then } S \mid \text{while } E \text{ do } S \mid S \ ; \ S$

array declaration (of size num)

pointer declaration

function declaration

integer literal

function call

character literal

array access

pointer dereference

- We want to write an ad-hoc translation scheme for type-checking

Type Expressions

- Using base types and type-constructors each expression in a program can be represented with a *type expression*

- A base type is a type expression
- A type name is a type expression
- A type constructor applied to type expressions is a type expression, for example:
  - array(I,T) : creates a type expression for an array of type T with index set I.
  - pointer(T) : creates a type expression of type pointer to type T.
  - function(T,T) : creates a type expression of type function from type T to type T.

Type Checking with this Tiny Language

- Static Type Checking (mainly about **operations + types** matching:
  - **Type matching for assignment** : The compiler checks whether the type expression of RHS matches the LHS of an assignment statement.
  - **Dereferencing checks.** The compiler checks that dereferencing is applied only to a pointer.
  - **Function call checks.** The compiler checks that a function (or procedure) is applied to the correct number and type of arguments.
  - **Special functionality**: arguments of modulo operation are both integers + more…
  - **Indexing checks.** The compiler checks that indexing is applied only to an array.

Semantic Actions for Type Checking: Declarations

$P \rightarrow D\ ;\ S$	
$D \rightarrow D\ ;\ D$	
$D \rightarrow$ id : T	{ addtype(id.entry, T.type); }
$T \rightarrow$ char	{ T.type $\leftarrow$ char; }
$T \rightarrow$ int	{ T.type $\leftarrow$ integer; }
$T \rightarrow$ array [num] of T_1	{ T.type $\leftarrow$ array(1...num.val, T_1.type); }
$T \rightarrow$ pointer T_1	{ T.type $\leftarrow$ pointer(T_1.type); }
$T \rightarrow$ function T_1 to T_2	{ T.type $\leftarrow$ function(T_1.type, T_2.type); }

Some Implementation Tricks for Type Expression Encoding

- Approach: Type Encodings
  - Basic types use a predefined encoding of the low-order bits
  - Basic Type Encoding

    boolean 0000

    char 0001

    integer 0002
  - The encoding of a type expression op(T) is obtained by concatenating the bits encoding op to the left of the encoding of T
  - Type Expression Encoding

    char 00 00 00 0001

    array(char) 00 00 01 0001

    ptr(array(char)) 00 10 01 0001

    ptr(ptr(array(char))) 10 10 01 0001

Semantic Actions for Type Checking: Expressions

$E \rightarrow$ literal	{ E.type $\leftarrow$ char; }
$E \rightarrow$ num	{ E.type $\leftarrow$ integer; }
$E \rightarrow$ id	{ E.type $\leftarrow$ lookup(id.entry); }
$E \rightarrow E_1$ mod E_2	{ if (E_1.type = integer and E_2.type = integer) then E.type $\leftarrow$ integer; else E.type $\leftarrow$ type-error; }
$E \rightarrow E_1$ [E_2]	{ if (E_2.type = integer and E_1.type = array(i,t)) then E.type $\leftarrow$ t; /* for some i and some t */ else E.type $\leftarrow$ type-error; }

Semantic Actions for Type Checking: Expressions

$E \rightarrow * E_1$	{ if (E_1.type = pointer(t)) /* for some t */ then E.type ← t; else E.type ← type-error; }
$E \rightarrow E_1 (E_2)$	{ if (E_2.type = s and E_1.type = function(s,t)) then E.type ← t; /* for some s and some t */ else E.type ← type-error; }
$E \rightarrow E_1 = E_2$	{ if (E_1.type == E_2.type) then E.type ← boolean; else E.type ← type-error; }

Semantic Actions for Type Checking: Statements

$S \rightarrow$ id := E	{ if (id.type = E.type) then S.type ← void; /* we could assign type of E here */ else S.type ← type-error; }
$S \rightarrow$ if E then S_1	{ if (E.type = boolean) then S.type ← S_1.type; else S.type ← type-error; }
$S \rightarrow$ while E do S_1	{ if (E.type = boolean) then S.type ← S_1.type; else S.type ← type-error; }
$S \rightarrow S_1 ; S_2$	{ if (S_1.type = void and S_2.type = void) then S.type ← void; else S.type ← type-error; }

Type Checking Example: Semantic Actions

(1) T.type = char

(2) assert(x not in symbol table)
add (x,char) to symbol table

(3) T.type = char

(4) T.type = pointer(T) = pointer(char)

(5) assert(y not in symbol table)
add (y,pointer(char)) to sym

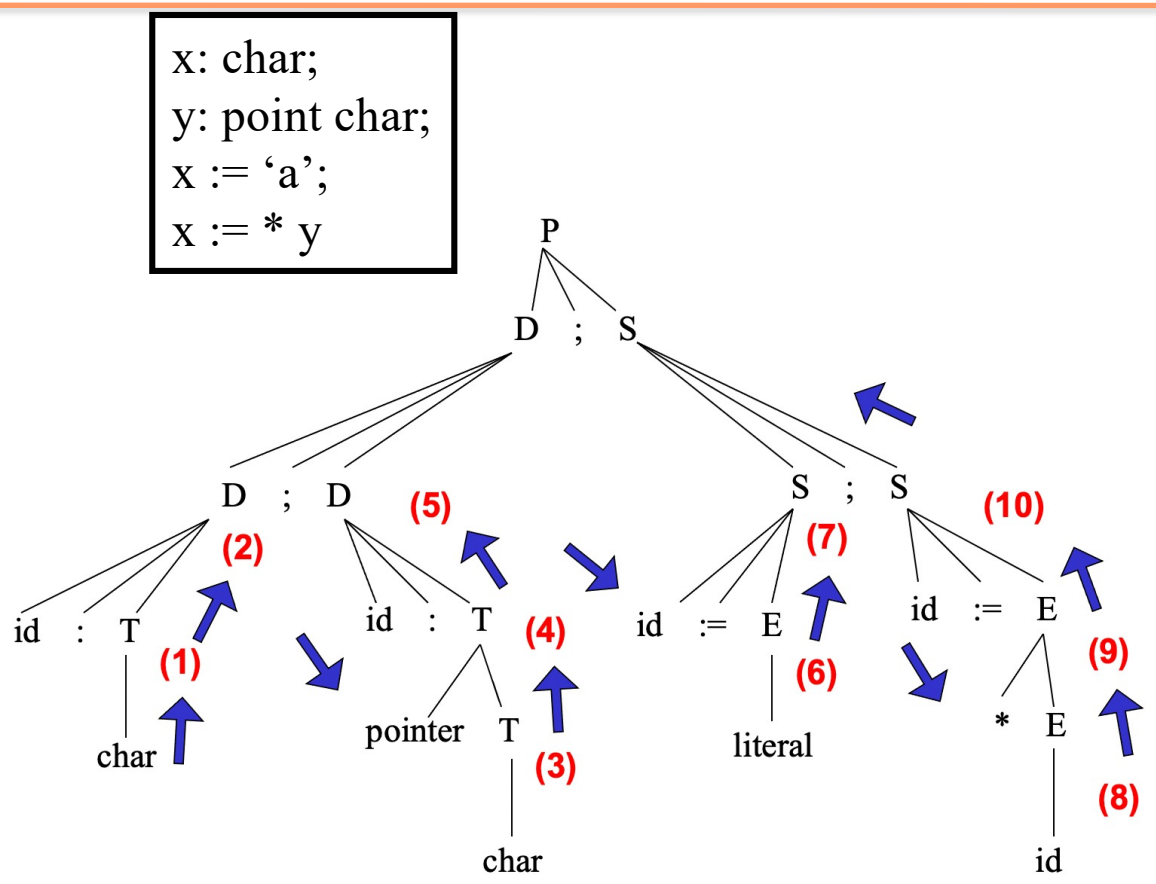
(6) E.type = char

(7) lookup(id); assert(id.exists);
assert(id.type==E.type)

(8) lookup(id); assert(id.exists);
E.type = id.type = pointer(char)

(9) assert(E.type == pointer(z) for some z); E.type = z = char

(10) lookup(id); assert(id.exists);
assert(id.type==E.type);



Common Type Checking

- Static Type Checking (mainly about **operations + types** matching:
  - **Type matching for assignment** : The compiler checks whether the type expression of RHS matches the LHS of an assignment statement.
  - **Dereferencing checks.** The compiler checks that dereferencing is applied only to a pointer.
  - **Function call checks.** The compiler checks that a function (or procedure) is applied to the correct number and type of arguments.
  - **Special functionality**: arguments of modulo operation are both integers + more…
  - **Indexing checks.** The compiler checks that indexing is applied only to an array.
  - **Type conversions.** Detection and application of implicit (explicit) type conversions.

Type Conversions

- Type Conversions can be
  - explicit (casts) if the programmer must write something to cause these conversions,
  - implicit (coercions) if done automatically by the compiler.
- From the point of view of type checking,
  - explicit conversions are just like function applications,
  - implicit conversions requires the type checker to detect that the two types are different and to insert the appropriate cast.
- Example (by including a **float** type):
  - $E \rightarrow E_1$ op E_2

{ if E_1.type = integer and E_2.type = integer, then E.type = integer;
else if E_1.type = integer and E_2.type = float, then E.type = float;
else if E_1.type = float and E_2.type = integer, then E.type = float;
else if E_1.type = float and E_2.type = float, then E.type = float;
else E.type = type-error;

Other Static Checking

- Flow-of-Control Check

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

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

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

Other Static Checking

- Uniqueness Checking

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

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

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