Lecture 7: Type Systems and Symbol Tables

CS 540

George Mason University
Static Analysis

Compilers examine code to find semantic problems.
- Easy: undeclared variables, tag matching
- Difficult: preventing execution errors

Essential Issues:
- Part I: Type checking
- Part II: Scope
- Part III: Symbol tables
Part I: Type Checking
Type Systems

• A type is a set of values and associated operations.
• A type system is a collection of rules for assigning type expressions to various parts of the program.
  – Impose constraints that help enforce correctness.
  – Provide a high-level interface for commonly used constructs (for example, arrays, records).
  – Make it possible to tailor computations to the type, increasing efficiency (for example, integer vs. real arithmetic).
  – Different languages have different type systems.
Inference Rules - Typechecking

• Static (compile time) and Dynamic (runtime).
• One responsibility of a compiler is to see that all symbols are used correctly (i.e. consistently with the type system) to prevent execution errors.
• Strong typing – All expressions are guaranteed to be type consistent although the type itself is not always known (may require additional runtime checking).
What are Execution Errors?

• Trapped errors – errors that cause a computation to stop immediately
  – Division by 0
  – Accessing illegal address

• Untrapped errors – errors that can go unnoticed for a while and then cause arbitrary behavior
  – Improperly using legal address (moving past end of array)
  – Jumping to wrong address (jumping to data location)

• A program fragment is safe if it does not cause untrapped errors to occur.
Typechecking

We need to be able to assign types to all expressions in a program and show that they are all being used correctly.

Input: \( x \times y + f(a,b) \)

- Are \( x \), \( y \) and \( f \) declared?
- Can \( x \) and \( y \) be multiplied together?
- What is the return type of function \( f \)?
- Does \( f \) have the right number and type of parameters?
- Can \( f \)'s return type be added to something?
Program Symbols

• User defines symbols with associated meanings. Must keep information around about these symbols:
  – Is the symbol declared?
  – Is the symbol visible at this point?
  – Is the symbol used correctly with respect to its declaration?
Using Syntax Directed Translation to process symbols

While parsing input program, need to:

• **Process declarations** for given symbols
  – Scope – what are the visible symbols in the current scope?
  – Type – what is the declared type of the symbol?

• **Lookup symbols used** in program to find current binding

• **Determine the type of the expressions** in the program
Syntax Directed Type Checking

Consider the following simple language

\[ P \rightarrow D \ S \]

\[ D \rightarrow \text{id: } T ; D \mid \varepsilon \]

\[ T \rightarrow \text{integer} \mid \text{float} \mid \text{array } [\text{num}] \text{ of } T \mid ^T \]

\[ S \rightarrow S ; S \mid \text{id := } E \]

\[ E \rightarrow \text{int literal} \mid \text{float literal} \mid \text{id} \mid E + E \mid E [ E ] \mid E^\]

How can we typecheck strings in this language?
Example of language:

\[ i: \text{integer}; j: \text{integer} \]

\[ i := i + 1; \]

\[ j := i + 1 \]
Processing Declarations

D \rightarrow \text{id : T } ; \text{D} \quad \{\text{insert(id.name, T.type);}\}

D \rightarrow \varepsilon

T \rightarrow \text{integer} \quad \{T.type = \text{integer};\}

T \rightarrow \text{float} \quad \{T.type = \text{float};\}

T \rightarrow \text{array [ num ] of } T_1 \quad \{T.type = \text{array}(T_1.type, \text{num}); \}

T \rightarrow ^{T_1} \quad \{T.type = \text{pointer}(T_1.type);\}

Accumulate information about the declared type

Put info into the symbol table
Can use Trees (or DAGs) to Represent Types

```
array[100] of ^(float)
```

Build data structures while we parse
Example

I: integer;
A: array[20] of integer;
B: array[20] of ^integer;
I := B[A[2]]^
Example

I: integer;
A: array[20] of integer;
B: array[20] of ^integer;
I := B[A[2]]^
Example

I: integer;
A: array[20] of integer;
B: array[20] of ^integer;
I := B[A[2]]^
Typechecking Expressions

E \rightarrow \text{int\_literal} \quad \{ \ E\.\text{type} := \text{integer}; \ \}
E \rightarrow \text{float\_literal} \quad \{ \ E\.\text{type} = \text{float}; \ \}
E \rightarrow \text{id} \quad \{ \ E\.\text{type} := \text{lookup(id.name)}; \ \}
E \rightarrow E_1 + E_2 \quad \{ \ \text{if (E}_1\.\text{type} = \text{integer} \ \& \ \text{E}_2\.\text{type} = \text{integer})
\text{then E\.type = integer;}
\text{else if (E}_1\.\text{type} = \text{float} \ \& \ \text{E}_2\.\text{type} = \text{float})
\text{then E\.type = float;}
\text{else type\_error();} \ \}
E \rightarrow E_1 [ E_2 ] \quad \{ \ \text{if (E}_1\.\text{type} = \text{array of T} \ \& \ \text{E}_2\.\text{type} = \text{integer})
\text{then E\.type = T; else …} \ \}
E \rightarrow E_1^\wedge \quad \{ \ \text{if (E}_1\.\text{type} = ^\wedge T)
\text{then E\.type = T; else …} \ \}

These rules define a type system for the language.
Example

I: integer;
A: array[20] of integer;
B: array[20] of ^integer;
I := B[A[2]]

\[
\begin{align*}
I & \text{ integer; } \\
A & \text{ array[20] of integer; } \\
B & \text{ array[20] of } ^\text{integer}; \\
I & := B[A[2]]
\end{align*}
\]
Example

I: integer;
A: array[20] of integer;
B: array[20] of ^integer;
I := B[A[2]]^
Example

I: integer;
A: array[20] of integer;
B: array[20] of ^integer;
I := A[B[2]]^
Typechecking Statements

\[ S \rightarrow S_1 ; S_1 \quad \{ \text{if } S_1.\text{type} = \text{void} & \ S_1.\text{type} = \text{void} \} \]
\[ \quad \text{then } S.\text{type} = \text{void}; \text{ else error(); } \} \]

\[ S \rightarrow \text{id} := E \quad \{ \text{if lookup(id.name)} = E.\text{type} \]
\[ \quad \text{then } S.\text{type} = \text{void}; \text{ else error(); } \} \]

\[ S \rightarrow \text{if E then } S_1 \quad \{ \text{if } E.\text{type} = \text{boolean} \text{ and } S_1.\text{type} = \text{void} \]
\[ \quad \text{then } S.\text{type} = \text{void}; \text{ else error(); } \} \]

In this case, we assume that statements do not have types
(not always the case).
Typechecking Statements

What if statements have types?

\[
S \rightarrow S_1 ; S_2 \quad \{ S\.type = S_2\.type; \}
\]

\[
S \rightarrow \text{id } := E \quad \{ \text{if lookup(id.name) } = E\.type \text{ then} \text{ S\.type = E\.type; else error();} \}
\]

\[
S \rightarrow \text{if E then } S_1 \text{ else } S_2 \quad \{ \text{if (E.type } = \text{ boolean } \& S_1\.type = S_2\.type) \text{ then} \text{ S\.type } = S_1\.type; \text{ else error();} \}
\]
Untyped languages

Single type that contains all values

• Ex:
  
  Lisp – program and data interchangeable
  Assembly languages – bit strings

• Checking typically done at runtime
Typed languages

- Variables have nontrivial types which limit the values that can be held.
- In most typed languages, new types can be defined using type operators.
- Much of the checking can be done at compile time!
- Different languages make different assumptions about type semantics.
Components of a Type System

• Base Types
• Compound/Constructed Types
• Type Equivalence
• Inference Rules (Typechecking)
• …

Different languages make different choices!
Base (built-in) types

• Numbers
  – Multiple – integer, floating point
  – precision
• Characters
• Booleans
Constructed Types

- Array
- String
- Enumerated types
- Record
- Pointer
- Classes (OO) and inheritance relationships
- Procedure/Functions
- ...
Type Equivalence

Two types: Structural and Name

Type A = Bool
Type B = Bool

• In Structural equivalence: Types A and B match because they are both boolean.

• In Name equivalivance: A and B don’t match because they have different names.
Implementing Structural Equivalence

To determine whether two types are structurally equivalent, traverse the types:

```java
boolean equiv(s, t) {
    if s and t are same basic type return true
    if s = array(s1, s2) and t is array(t1, t2)
        return equiv(s1, t1) & equiv(s2, t2)
    if s = pointer(s1) and t = pointer(t1)
        return equiv(s1, t1)
    ...
    return false;
}
```
Other Practical Type System Issues

• Implicit versus explicit type conversions
  – Explicit $\Rightarrow$ user indicates (Ada)
  – Implicit $\Rightarrow$ built-in (C int/char) -- coercions

• Overloading – meaning based on context
  – Built-in
  – Extracting meaning – parameters/context

• Objects (inheritance)

• Polymorphism
OO Languages

• Data is organized into classes and sub-classes
• Top level is class of all objects
• Objects at any level inherit the attributes (data, functions) of objects higher up in the hierarchy. The subclass has a larger set of properties than the class. Subclasses can override behavior inherited from parent classes. (But cannot revise private data elements from a parent).
class A {
    public: A() {cout << "Creating A\n"; }
    W() {cout << "W in A\n"; }
};
class B: public A {
    public: B() {cout << "Creating B\n"; }
    S() {cout << "S in B\n"; }
};
class C: public A {
    public: C() {cout << "Creating C\n"; }
    Y() {cout << "Y in C\n"; }
};
class D: public C {
    public: D() {cout << "Creating D\n"; }
    S() {cout << "S in D\n"; }
};
The Code:

```
B b;
D d;

b.W();
b.S();
d.W();
d.Y();
d.S();
```

Output:

```
Creating A
Creating B
Creating C
Creating D
W in A
S in B
W in A
Y in C
S in D
```
OO Principle of Substitutability

• Subclasses possess all data areas associated with parent classes
• Subclasses implement (through inheritance) at least all functionality defined for the parent class

If we have two classes, A and B, such that class B is a subclass of A (perhaps several times removed), it should be possible to substitute instances of class B for instances of class A in any situation with no observable effect.
Typechecking OO languages

• Without inheritance, the task would be relatively simple (similar to records)

• Difficulties:
  – Method overriding
  – When can super/sub types be used? Consider function \( f : A \rightarrow B \)
    • Actual parameter of type A or subtype of A
    • Return type B or supertype of B
  – Multiple inheritance
Function parameters

- Function parameters make typechecking more difficult

```
procedure mlist(lptr: link; procedure p)
    while lptr <> nil begin
        p(lptr);
        lptr = lptr → next;
    end
end
```
Polymorphism

- Functions – statements in body can be executed on arguments with different type – common in OO languages because of inheritance
- Ex: Python for determining the length of a list
  
  ```python
  def size (lis):
      if null(lis):
          return 0
      else:
          return size(lis[1:]) + 1;
  
  size(['sun','mon','tue'])
  size([10,11,12])
  size(A)
  ```
def size (lis):
    if null(lis):
        return 0
    else:
        return size(lis[1:])+1;

Goal: determine a type for size so we can typecheck the calls.

Greek symbols are type variables.

Fig 6.30 of your text
Type Inferencing

def size(lis):
    if null(lis):
        return 0
    else:
        return size(lis[1:])+1;

Built-in language constructs and functions provide clues.

Given what we have in the table, we now know that \( \text{list}(\alpha_n) = \beta \)

Fig 6.30 of your text
Type Inferencing

def size (lis):
    if null(lis):
        return 0
    else:
        return size(lis[1:]) + 1;

\( \alpha_i = \text{int} \)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>( \beta \rightarrow \gamma )</td>
</tr>
<tr>
<td>lis</td>
<td>( \beta \ (\text{list}(\alpha_n)) )</td>
</tr>
<tr>
<td>if</td>
<td>( \text{bool} \ \times \ \alpha_i \ \times \ \alpha_i \rightarrow \alpha_i )</td>
</tr>
<tr>
<td>null</td>
<td>( \text{list}(\alpha_n) \rightarrow \text{bool} )</td>
</tr>
<tr>
<td>null(lis)</td>
<td>( \text{bool} )</td>
</tr>
<tr>
<td>0</td>
<td>( \text{int} )</td>
</tr>
<tr>
<td>+</td>
<td>( \text{int} \ \times \ \text{int} \rightarrow \text{int} )</td>
</tr>
<tr>
<td>lis[1:]</td>
<td>( \text{list}(\alpha_n) )</td>
</tr>
</tbody>
</table>

Fig 6.30 of your text
def size (lis):
    if null(lis):
        return 0
    else:
        return size(lis[1:]) + 1;

γ = int

All of this tells us that
size: list(α) → int
(in other words, maps from anything with type list to type integer)

Fig 6.30 of your text
Formalizing Type Systems

• Mathematical characterizations of the type system – Type soundness theorems.
• Requires formalization of language syntax, static scoping rules and semantics.
• Formalization of type rules

• [Link](http://research.microsoft.com/users/luca/Papers/TypeSystems.pdf)
Part II: Scope
Scope

In most languages, a complete program will contain several different **namespaces** or **scopes**.

Different languages have different rules for namespace definition.
Fortran 77 Name Space

Global scope holds procedure names and common block names. Procedures have local variables, parameters, labels and can import common blocks.
Scheme Name Space

- All objects (built-in and user-defined) reside in single global namespace
- ‘let’ expressions create nested lexical scopes
C Name Space

- Global scope holds variables and functions
- No function nesting
- Block level scope introduces variables and labels
- File level scope with static variables that are not visible outside the file (global otherwise)
Java Name Space

- Limited global name space with only public classes
- Fields and methods in a public class can be public visible to classes in other packages
- Fields and methods in a class are visible to all classes in the same package unless declared private
- Class variables visible to all objects of the same class.
Scope

Each scope maps a set of variables to a set of meanings.

The scope of a variable declaration is the part of the program where that variable is visible.
Referencing Environment

The referencing environment at a particular location in source code is the set of variables that are visible at that point.

- A variable is **local** to a procedure if the declaration occurs in that procedure.
- A variable is **non-local** to a procedure if it is visible inside the procedure but is not declared inside that procedure.
- A variable is **global** if it occurs in the outermost scope (special case of non-local).
Types of Scoping

- Static – scope of a variable determined from the source code.
  - “Most Closely Nested”
  - Used by most languages

- Dynamic – current call tree determines the relevant declaration of a variable use.
Static: Most Closely Nested Rule

The scope of a particular declaration is given by the most closely nested rule

- The scope of a variable declared in block B, includes B.
- If x is not declared in block B, then an occurrence of x in B is in the scope of a declaration of x in some enclosing block A, such that A has a declaration of x and A is more closely nested around B than any other block with a declaration of x.
Example Program: Static

Program `main`;
  `a, b, c`: real;
  procedure `sub1`(a: real);
    `d`: int;
    procedure `sub2`(c: int);
      `d`: real;
      body of `sub2`
    body of `sub2`
  procedure `sub3`(a: int)
    body of `sub3`
  body of `sub1`
body of `main`

What is visible at this point (globally)?
Example Program: Static

Program main;
  a, b, c: real;

  procedure sub1(a: real);
    d: int;
    procedure sub2(c: int);
      d: real;
    body of sub2
  body of sub2

  procedure sub3(a: int)
    body of sub3

body of sub1

body of main

What is visible at this point (sub1)?
Example Program: Static

Program main;
  a, b, c: real;

procedure sub1(a: real);
  d: int;

procedure sub2(c: int);
  d: real;

body of sub2

procedure sub3(a: int)
body of sub3

body of sub1

body of main

What is visible at this point (sub3)?
Example Program: Static

Program main;
a, b, c: real;

procedure sub1(a: real);
d: int;

procedure sub2(c: int);
d: real;

body of sub2

procedure sub3(a: int)
body of sub3

body of sub1

body of main

What is visible at this point (sub2)?
Dynamic Scope

• Based on calling sequences of program units, not their textual layout (temporal versus spatial)

• References to variables are connected to declarations by searching the chain of subprogram calls (runtime stack) that forced execution to this point
Scope Example

MAIN
- declaration of x
  SUB1
    - declaration of x -
    ...
    call SUB2
    ...
  SUB2
    ...
    - reference to x -
    ...
...
call SUB1
...

MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses x

Which x??
Scope Example

MAIN
- declaration of x
  SUB1
  - declaration of x -
  ...
  call SUB2
  ...

SUB2
  ...
  - reference to x -
  ...

... call SUB1
...

MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses x

For static scoping, it is main’s x
Scope Example

- In a dynamic-scoped language, the referencing environment is the local variables plus all visible variables in all active subprograms.
- A subprogram is active if its execution has begun but has not yet terminated.
Scope Example

MAIN
(x)

SUB1
(x)

SUB2

- declaration of x
  SUB1
  - declaration of x -
  ...
  call SUB2
  ...

SUB2
...
- reference to x -
...

... call SUB1 ...

MAIN calls SUB1
SUB1 calls SUB2
SUB2 uses x

For dynamic scoping, it is sub1’s x
Dynamic Scoping

• Evaluation of Dynamic Scoping:
  – Advantage: convenience (easy to implement)
  – Disadvantage: poor readability, unbounded search time
Part III: Symbol Tables
Symbol Table

• Primary data structure inside a compiler.
• Stores information about the symbols in the input program including:
  – Type (or class)
  – Size (if not implied by type)
  – Scope
• Scope represented explicitly or implicitly (based on table structure).
• Classes can also be represented by structure – one difference = information about classes must persist after have left scope.
• Used in all phases of the compiler.
Symbol Table Object

Symbol table functions are called during parsing:

• Insert(x) – *A new symbol is defined.*
• Delete(x) – *The lifetime of a symbol ends.*
• Lookup(x) – *A symbol is used.*
• EnterScope(s) – *A new scope is entered.*
• ExitScope(s) – *A scope is left.*
Scope and Parsing

\[
\text{func\_decl} : \text{FUNCTION NAME} \quad \{\text{EnterScope($2);}\}
\]
\[
\text{parameter decls stmts} ; \quad \{\text{ExitScope($2);}\}
\]
\[
\text{decl} : \quad \text{name `:` type} \quad \{\text{Insert($1,$3);}\}
\]

\[
\ldots
\]
\[
\text{statements:} \quad \text{id} := \text{expression} \quad \{\text{lookup($1);}\}
\]
\[
\ldots
\]
\[
\text{expression:} \quad \ldots
\]
\[
\text{id} \quad \{\text{lookup($1);}\}
\]

Note: This is a greatly simplified grammar including only the symbol table relevant productions.
Symbol Table Implementation

- Variety of choices, including arrays, lists, trees, heaps, hash tables, ...
- Different structures may be used for local tables versus tables representing scope.
Example Implementation

• Local level – within a scope, use a table or linked list.

• Global – each scope is represented as a structure that points at –
  – Its local symbols
  – The scopes that it encloses
  – Its enclosing scope

\[ \text{A tree?} \]
Implementing the table

- Need variable CS for current scope
- *EnterScope* – creates a new record that is a child of the current scope. This scope has new empty local table. Set CS to this record.
- *Insert* – add a new entry to the local table of CS
- *Lookup* – Search local table of CS. If not found, check the enclosing scope. Continue checking enclosing scopes until found or until run out of scopes.
Example Program

Program main;
  a, b, c: real;

procedure sub1(a: real);
  d: int;

procedure sub2(c: int);
  d: real;
body of sub2

procedure sub3(a: int)
body of sub3

body of sub1
body of main

Main

sub1

sub2

sub3

a, b, c

a, d

sub1

sub2

sub3

c, d

a
Implementing the table

We can use a stack instead!!!

• **EnterScope** – creates a new record that is a child of the current scope. This scope has new empty local table. Set CS to this record ➔ **PUSH**

• **ExitScope** – set CS to parent of current scope. Update tables ➔ **POP**

• **Insert** – add a new entry to the local table of CS

• **Lookup** – Search local table of CS. If not found, check the enclosing scope. Continue checking enclosing scopes until found or until run out of scopes.
Example Program – As we compile …

Program main;
  a, b, c: real;

  procedure sub1(a: real);
    d: int;

      procedure sub2(c: int);
        d: real;

      body of sub2

      procedure sub3(a: int)

        body of sub3

  body of sub1

body of main
Example Program

Program main;
  a,b,c: real;

  procedure sub1(a: real);
    d: int;
    
    procedure sub2(c: int);
      d: real;
      
      body of sub2
    
    procedure sub3(a:int)
      body of sub3
    
  body of sub1

body of main
Example Program

Program main;
  a, b, c: real;
  procedure sub1(a: real);
    d: int;
    procedure sub2(c: int);
      d: real;
      body of sub2
    body of sub2
  procedure sub3(a: int)
    body of sub3
  body of sub1
body of main
Example Program

Program main;
    a, b, c: real;

    procedure sub1(a: real);
        d: int;

        procedure sub2(c: int);
            d: real;
        body of sub2

        procedure sub3(a: int)
        body of sub3
    body of sub1

    body of main
Example Program

Program main;
a, b, c: real;

procedure sub1(a: real);
d: int;

procedure sub2(c: int);
d: real;
body of sub2

procedure sub3(a: int)
body of sub3

body of sub1

body of main
Example Program

Program main;
    a, b, c: real;

    procedure sub1(a: real);
        d: int;

        procedure sub2(c: int);
            d: real;
        body of sub2

        procedure sub3(a: int)
            body of sub3

    body of sub1

body of main