Programming Languages & Translators

SEMANTIC ANALYSIS

Baishakhi Ray

Fall 2018

These slides are motivated from Prof. Alex Aiken and Prof. Stephen Edward

The Compiler So Far

- Lexical analysis
  - Detects inputs with illegal tokens
- Parsing
  - Detects inputs with ill-formed parse trees
- Semantic analysis
  - Last "front end" phase
  - Catches all remaining errors

What's Wrong With This?

`a + f(b, c)`

What's Wrong With This?

`a + f(b, c)`

Is a defined?

Is f defined?

Are b and c defined?

Is f a function of two arguments?

Can you add whatever a is to whatever f returns?

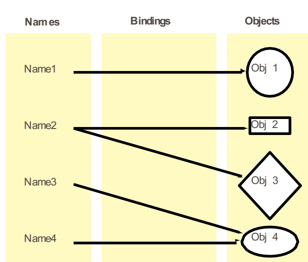
Does f accept whatever b and c are?

Scope questions   Type questions

parsing alone cannot answer these question.

Scope

- The scope of an identifier is the portion of a program in which that identifier is accessible.
- The same identifier may refer to different things in different parts of the program.
  - Different scopes for same name don't overlap.
- An identifier may have restricted scope.



Static Vs. Dynamic Scoping

- Most modern languages have static scope
  - Scope depends only on the program text, not runtime behavior
  - Most modern languages use static scoping. Easier to understand, harder to break programs.
- A few languages are dynamically scoped
  - Scope depends on execution of the program
  - Lisp, SNOBOL (Lisp has changed to mostly static scoping)
  - Advantage of dynamic scoping: ability to change environment.
  - A way to surreptitiously pass additional parameters.

Basic Static Scope in C, C++, Java, etc.

A name begins life where it is declared and ends at the end of its block.

From the CLRM, "The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block."

```
void foo()
{
   int x;

}
```

Hiding a Definition

Nested scopes can hide earlier definitions, giving a hole.

From the CLRM, "If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block."

```
void foo()
{
  int x;

  while ( a < 10 ) {
    int x;
  }

}
```

Dynamic Definitions in TEX

```
% \x, \y undefined
{
  % \x, \y undefined
  \def \x 1
  % \x defined, \y undefined

  \ifnum \a < 5
     \def \y 2
  \fi

  % \x defined, \y may be undefined
}
% \x, \y undefined
```

Open vs. Closed Scopes

- An *open scope* begins life including the symbols in its outer scope.
- Example: blocks in Java

```
{
 int x;
 for (;;) {
    /* x visible here */
 }
}
```

- A *closed scope* begins life devoid of symbols. Example: structures in C.

```
struct foo { int x; float y; }
```

Symbol Tables

- A symbol table is a data structure that tracks the current bindings of identifiers
- Can be implemented as a stack
- Operations
  - add_symbol(x) push x and associated info, such as x's type, on the stack
  - find_symbol(x) search stack, starting from top, for x. Return first x found or NULL if none found
  - remove_symbol() pop the stack when out of scope
- Limitation:
  - What if two identical objects are defined in the same scope multiple times.
  - Eg: foo(int x, int x)

Advanced Symbol Table

- enter_scope() start a new nested scope
- find_symbol(x) finds current x (or null)
- add_symbol(x) add a symbol x to the table
- check_scope(x) true if x defined in current scope
- exit_scope() exit current scope

Types

- What is a type?
  - The notion varies from language to language
- Consensus
  - A set of values
  - A set of operations on those values
- Classes are one instantiation of the modern notion of type

Why Do We Need Type Systems?

- Consider the assembly language fragment
  add $r1, $r2, $r3
- What are the types of $r1, $r2, $r3?
- 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
  - But both have the same assembly language implementation!

Type Systems

- A language's type system specifies which operations are valid for which types
- The goal of type checking is to ensure that operations are used with the correct types
  - Enforces intended interpretation of values, because nothing else will!
- Three kinds of languages:
  - Statically typed: All or almost all checking of types is done as part of compilation (C, Java)
  - Dynamically typed: Almost all checking of types is done as part of program execution (Python)
  - Untyped: No type checking (machine code)

Static vs. Dynamic Typing

- 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 difficult within a static type system
- In practice
  - code written in statically typed languages usually has an escape mechanism - Unsafe casts in C, Java
  - Some dynamically typed languages support "pragmas" or "advice" - i.e., type declarations.

Type Checking and Type Inference

- Type Checking is the process of verifying fully typed programs
- Type Inference is the process of filling in missing type information
- The two are different, but the terms are often used interchangeably
- Rules of Inference
  - We have seen two examples of formal notation specifying parts of a compiler : Regular expressions, Context-free grammars
  - The appropriate formalism for type checking is logical rules of inference

Why Rules of Inference?

- Inference rules have the form If Hypothesis is true, then Conclusion is true
- Type checking computes via reasoning

  If E1 and E2 have certain types, then E3 has a certain type
- Rules of inference are a compact notation for "If-Then" statements

From English to an Inference Rule

- The notation is easy to read with practice
- Start with a simplified system and gradually add features
- Building blocks
  - Symbol ∧ is "and"
  - Symbol ⇒ is "if-then"
  - x:T is "x has type T"
- If e_1 has type Int and e_2 has type Int, then $e_1 + e_2$ has type Int
  - (e1 has type Int ∧ e2 has type Int) ⇒ e1 + e2 has type Int
  - (e1: Int ∧ e2: Int) ⇒ e1 + e2: Int
  - It is a special case of $Hypothesis_1 \wedge \ldots \wedge Hypothesis_n \Rightarrow Conclusion$ (This is an inference rule).

Notation for Inference Rules

- By tradition inference rules are written

$$\frac{\vdash Hypothesis \ \ldots \vdash Hypothesis}{\vdash Conclusion}$$

⊢ e:T means "it is provable that e is of type T

Two Rules

$$\frac{\vdash i \text{ is an integer literal}}{\vdash i: Int} \ [Int]$$

$$\frac{\vdash e1: Int \quad \vdash e2: Int}{\vdash e1 + e2: Int} \ [Add]$$

$$\frac{\vdash e: Bool}{\vdash !e: Bool} \ [Not]$$

- These rules give templates describing how to type integers and + expressions
- By filling in the templates, we can produce complete typings for expressions
- Example: 1 + 2?

Type Checking Proofs

- Type checking proves facts e: T
  - Proof is on the structure of the AST
  - Proof has the shape of the AST
  - One type rule is used for each AST node
- In the type rule used for a node e:
  - Hypotheses are the proofs of types of e's sub-expressions
  - Conclusion is the type of e
- Types are computed in a bottom-up pass over the AST

A Problem

- What is the type of a variable reference?
- $\frac{x \text{ is a variable}}{\vdash x: ?}$

- The local, structural rule does not carry enough information to give x a type.

A solution

- Put more information in the rules!
- A type environment gives types for free variables
  - A type environment is a function from ObjectIdentifiers to Types
  - A variable is free in an expression if it is not defined within the expression

- Type Environments
  - Let O be a function from ObjectIdentifiers to Types
    The sentence O ⊢ e: T
    is read: Under the assumption that variables have the types given by O, it is provable that the expression e has the type T
  - $\frac{O(x) = T}{\vdash x: T}$

Implementing Type Checking

$$\frac{O, M, C \vdash e1 : Int \quad O, M, C \vdash e2 : Int}{O, M, C \vdash e1 + e2 : Int}$$

```
TypeCheck(Environment, e1 + e2) = {
T1 = TypeCheck(Environment, e1);
T2 = TypeCheck(Environment, e2);
Check T1 == T2 == Int;
return Int; }
```

Binding Time

When are bindings created and destroyed?

Binding Time

When a name is connected to an object.

Bound when	Examples
language designed	if else
language implemented	datatype widths
Program written	foo bar
compiled	static addresses, code
linked	relative addresses shared objects
loaded	
run	heap-allocated objects

Binding Time and Efficiency

Earlier binding time ⇒ more efficiency, less flexibility

Compiled code more efficient than interpreted because most decisions about what to execute made beforehand.

```
switch (statement) {

case add:
   r = a + b;
   break;

case sub:
   r = a - b;
   break;

   /* ... */
}
```

```
add %o1, %o2, %o3
```

Binding Time and Efficiency

Dynamic method dispatch in OO languages:

```
class Box : Shape {
  public void draw() { ... }
}

class Circle : Shape {
  public void draw() { ... }
}

Shape s;
s.draw(); /* Bound at run time */
```

Binding Time and Efficiency

Interpreters better if language has the ability to create new programs on-the-fly.

Example: Ousterhout's Tcl language.

Scripting language originally interpreted, later byte-compiled.

Everything's a string.

```
set a 1
set b 2
puts "$a + $b = [expr $a + $b]"
```

Static Semantic Analysis

How do we validate names, scope, and types?

Static Semantic Analysis

Lexical analysis: Each token is valid?

```
if i 3 "This"              /* valid Java tokens */
#a1123                     /* not a token */
```

Syntactic analysis: Tokens appear in the correct order?

```
for ( i = 1 ; i < 5 ; i++ ) 3 + "foo"; /* valid Java syntax */
for break                              /* invalid syntax */
```

Semantic analysis: Names used correctly? Types consistent?

```
int v = 42 + 13;     /* valid in Java (if v is new) */
return f + f(3);     /* invalid */
```

What To Check

Examples from Java:

Verify names are defined and are of the right type.

```
int i = 5;
int a = z;      /* Error: cannot find symbol */
int b = i[3];   /* Error: array required, but int found */
```

Verify the type of each expression is consistent.

```
int j = i + 53;
int k = 3 + "hello";          /* Error: incompatible types */
int l = k(42);                /* Error: k is not a method */
if ("Hello") return 5;        /* Error: incompatible types */
String s = "Hello";
int m = s;                    /* Error: incompatible types */
```

How To Check Expressions: Depth-first AST Walk

Checking function: environment → node → type

```
  1 - 5                 1 + "Hello"

    -                       +
   / \                     / \
  1   5                   1   "Hello"
```

check(−)
- check(1) = int
- check(5) = int
- Success: int − int = int

check(+)
- check(1) = int
- check("Hello") = string
- FAIL: Can't add int and string

Ask yourself: at each kind of node, what must be true about the nodes below it? What is the type of the node?

How To Check: Symbols

Checking function: environment → node → type

```
  1 + a

    +
   / \
  1   a
```

check(+)
- check(1) = int
- check(a) = int
- Success: int + int = int

The key operation: determining the type of a symbol when it is encountered.

The environment provides a "symbol table" that holds information about each in-scope symbol.

A Static Semantic Checking Function

A big function: "check: ast → sast"

Converts a raw AST to a "semantically checked AST"

Names and types resolved

```
AST
type expression =
        IntConst of int
      | Id of string
      | Call of string * expression list
      | ...
```

→

```
SAST
type expr_detail =
        IntConst of int
      | Id of variable_decl
      | Call of function_decl * expression list
      | ...type expression = expr_detail * Type.t
```

The Type of Types

Need an OCaml type to represent the type of something in your language.

An example for a language with integer, structures, arrays, and exceptions:

```
type t = (* can't call it "type" since that's reserved *)
    Void
  | Int
  | Struct of string * ((string * t) array)   (* name, fields *)
  | Array of t * int.                         (* type, size *)
  | Exception of string
```

Translation Environments

Whether an expression/statement/function is correct depends on its context. Represent this as an object with named fields since you will invariably have to extend it.

An environment type for a C-like language:

```
type translation_environment ={
    scope : symbol_table;                    (* symbol table for vars *)

    return_type : Types.t;                   (* Function's return type *)
    in_switch : bool;                        (* if we are in a switch stmt *)
    case_labels : Big_int.big_int list ref;  (* known case labels *)
    break_label :label option;               (* when break makes sense *)
    continue_label : label option;           (* when continue makes sense *)
    exception_scope : exception_scope;       (* sym tab for exceptions *)
    labels : label list ref;                 (* labels on statements *)
    forward_gotos : label list ref;          (* forward goto destinations *)
  }
```

A Symbol Table

Basic operation is string → type. Map or hash could do this, but a list is fine.

```
type symbol_table ={
  parent : symbol_table option;
  variables : variable_decl list
}

let rec find_variable (scope : symbol_table) name =
  try
     List.find (fun (s, _, _, _) -> s = name) scope.variables
  with Not_found ->
    match scope.parent with
      Some(parent) -> find_variable parent name
    | _ -> raise Not_found
```

Checking Expressions: Literals and Identifiers

```
(* Information about where we are *)
type translation_environment ={
   scope : symbol_table;
}

let rec expr env =function

    (* An integer constant: convert and return Int type *)
    Ast.IntConst(v) -> Sast.IntConst(v), Types.Int

    (* An identifier: verify it is in scope and return its type *)
  | Ast.Id(vname) ->
     let vdecl =try
       find_variable env.scope vname (* locate a variable by name *)
     with Not_found ->
        raise (Error("undeclared identifier " ^ vname))
     in
     let (_, typ) = vdecl in (* get the variable's type *)
     Sast.Id(vdecl), typ

  | ...
```

Checking Expressions: Binary Operators

```
(* let rec expr env = function *)

| A.BinOp(e1, op, e2) ->
   let e1 = expr env e1    (* Check left and right children *)
   and e2 = expr env e2 in

   let _, t1 = e1          (* Get the type of each child *)
   and _, t2 = e2 in

   if op <> Ast.Equal && op <> Ast.NotEqual then
     (* Most operators require both left and right to be integer *)
     (require_integer e1 "Left operand must be integer";
      require_integer e2 "Right operand must be integer")
   else
     if not (weak_eq_type t1 t2) then
       (* Equality operators just require types to be "close" *)
       error ("Type mismatch in comparison: left is " ^
              Printer.string_of_sast_type t1 ^ "\" right is \"" ^
              Printer.string_of_sast_type t2 ^ "\""
             ) loc;

   Sast.BinOp(e1, op, e2), Types.Int (* Success: result is int *)
```

Checking Statements: Expressions, If

```
let rec stmt env = function

    (* Expression statement: just check the expression *)
    Ast.Expression(e) -> Sast.Expression(expr env e)

    (* If statement: verify the predicate is integer *)
  | Ast.If(e, s1, s2) ->

      let e = check_expr env e in (* Check the predicate *)
      require_integer e "Predicate of if must be integer";

      Sast.If(e, stmt env s1, stmt env s2) (* Check then, else *)
```

Checking Statements: Declarations

```
(* let rec stmt env = function *)

| A.Local(vdecl) ->
    let decl, (init, _) = check_local vdecl (* already declared? *)
    in

   (* side-effect: add variable to the environment *)
    env.scope.S.variables <- decl :: env.scope.S.variables;

    init (* initialization statements, if any *)
```

Checking Statements: Blocks

```
(* let rec stmt env = function *)

| A.Block(sl) ->

    (* New scopes: parent is the existing scope, start out empty *)

    let scope' = { S.parent = Some(env.scope); S.variables = [] }
    and exceptions' =
       { excep_parent = Some(env.exception_scope); exceptions = [] }
    in

    (* New environment: same, but with new symbol tables *)
    let env' = { env with scope = scope';
                 exception_scope = exceptions' } in

    (* Check all the statements in the block *)

    let sl = List.map (fun s -> stmt env' s) sl in
    scope'.S.variables <-
       List.rev scope'.S.variables;   (* side-effect *)

    Sast.Block(scope', sl) (* Success: return block with symbols *)
```