CS143: Semantic Analysis

David L. Dill
Stanford University
Semantic Analysis

• Introduction
• Scoped Symbol Tables
• Type Checking
Introduction
“Run-time” vs “Compile-time”

“Run-time” — happens when program is executed. “Dynamic”

“Compile-time” — happens when program is being compiled. “Static”
Better diagram

Front End → IR → IR Optimization → IR → Back End

↑ target independent optimizations

↑ target dependent optimizations
Front End

Input (string) → Lexical analysis → Syntactic analysis (Parsing) → Semantic Analysis

Symbol table
Semantic Analysis

"Meaning" of program ← whatever that means

Machine-independent processing

"Static" - compiler can do it.
Not context-free - parser can't do it.
Semantic Analysis

- Process and store declarations
  
  Bind name to description
  
  Classes, attributes, methods, formal parameters, etc.

  Look up correct description for name

Errors: "undefined variable/class/etc."
Semantic Analysis

Type checking

Inference - find type of expression

Checking - do types match?

Let \( x : \text{Int} \) in

Let \( y : \text{String} \leftarrow x + 1 \)

type checking

\( \text{int} \) \( \text{int} \) \( \text{int} \) \( \text{int} \)

\( \rightarrow \) type mismatch!
Semantic Analysis

Miscellaneous checks

Multiple definitions (if not allowed)
Assignable values (true ↔ false)
Proper inheritance (e.g., no cycles)
Proper use of reserved words
etc.

Highly language-dependent.
Scoped Symbol Tables
Declaration

Introduce a name for something.

Class Foo {
    f(x: Int) {
        x + 1;
    };
    a: Int <- 0;
}

Foo -> class def
f -> method def
x -> formal def

a -> attribute def
Definitions (not complete)

Class — Parent class
Features: attribute & method declarations (private symbol table) etc.

Attribute — type, initialization
Method — parameters, return type, body, etc.
Def: Name space - name → definition mapping (i.e. "symbol table")

"Separate name spaces" - different symbol tables.
Choose based on identifiers context
Namespaces in CooL

Class names are in their own namespace

Every class has a separate name space
methods, attributes are separate within
the class.
Def: Scope of a declaration - region of a program where declaration is accessible.

What declaration does ID refer to?
Def: Scope of a declaration - region of a program where declaration is accessible.

```
let x: Int <- 0 in
{ x;
  let m: Int <- 1 in
  { x;
    x;
  } }  # nested scope
```
Example

Let \( x : \text{Int} \leftarrow 0 \) in \( e \)

- declare \( x \) to be of type \( \text{Int} \) here

After \( e \), decl of \( x \) is undone. Symbol table is restored to state before let.

That definition applies in \( e \) (unless there is an inner let \( x \))
Symbol Table (symtab.h)

logically, it is a stack of tables

enterScope() — start a new scope

exitScope() — restore symbol table to state just before matching enterScope
Symbol Table (symtab.h)

addid(s, d) — add a declaration d for symbol s to symtab.

lookup(s) — search for s in all enclosing scopes, starting with the innermost.

probe(s) — search for s in current scope only.
Tree Traversal

\( \text{visit} (n) \)

*do stuff*

\( \text{visit} (\text{first child}) \)

*do more stuff*

\( \text{visit} (\text{next child}) \)

*do more stuff*

*do final stuff*
Tree Traversal

visit (n)
do stuff
visit (first child)
do more stuff
visit (next child)
do more stuff
visit (final child)
do final stuff
Tree Traversal

let

id type init expr

enter a scope
process type
add id/type to symtab
process expr
exit the scope
**Def.** Forward reference — use of a declaration appears earlier than the declaration itself.

**Cool:** Class names, features can be used before declaration.

Enables recursive data structures, methods, formals, let variables must be declared before uses.
Handling Class names, Features

Multiple Passes

Pass 1: Collect declarations
Use partial information if necessary

Pass 2: Processing that needs declarations

Use as many passes as is convenient.
Announcements

• Schedule revisions
  • Due dates pushed back a little
  • WA3/4 merged

• PA3 is hard. We’re going to write up some advice this afternoon.
  • Start: Figure out how to find classes
  • Understand symbol table (and test it).
Type Checking
"Type" is very programming language dependent

Consensus: A type is
- A set of values
- A set of operations on those values.

E.g. Classes
Goal of Type System

Machines think everything is a byte

No enforcement of reasonable operations for data.

No knowledge of size of data

Behavior of operations cannot change based on type of data.
Programming language Types

Reduce application of operations to inappropriate data.

Indicate size of data (e.g., for copying).

Allow behavior of operation to depend on data.
Dynamic type system – types are checked when operation is applied.

Run-time error reporting

Examples: Javascript, Python, LISP

Static typing – types are checked by compiler.

Examples: C++, Java (mostly), COOL
Dynamic types -

+ Fewer compile-time errors
- More run-time errors
  (Less reliable code, more debugging)
- Greater runtime overhead in time and space.

Static typing - swap +1 -
Def: Type Inference — fill in missing type information

If \( x: \text{Int}, y: \text{Int}, \) then \((x+y): \text{Int}\)

Def: Type checking — verify that all values are consistent with types.
Types in Cool

Class names

SELF_TYPE

User declares types of identifiers.

Compiler infers types of all expressions
Inference Rules

Formal notation borrowed from logic.

Often not used in language definitions.

No mainstream automated tools.
Soundness

No run-time type errors.

Whenever $\Gamma \vdash e : T$

then run-time value of $e$ has type $T$. 
Conventional Notation

$\vdash e_1 : \text{Int}$  $\vdash e_2 : \text{Int}$

$\vdash e_1 + e_2 : \text{Int}$

If $e_1$ is of type Int and $e_2$ is of type Int, then $e_1 + e_2$ is of type Int.
Typing Complex Expressions

\[
\begin{align*}
\text{\underline{i is an int literal}} & \quad \vdash \text{i : Int} \\
\text{\underline{e_1 : Int \ e_2 : Int}} & \quad \vdash \text{e_1 + e_2 : Int} \\
\text{\underline{1 is an int literal}} & \quad \vdash \text{1 : Int} \\
\text{\underline{2 is an int literal}} & \quad \vdash \text{2 : Int} \\
\hline
\vdash \text{1 + 2 : Int}
\end{align*}
\]
Code to Check Type Rules

Hand-written

One type rule per AST node

Premises are types of subexpressions

Types are computed bottom-up.
Rules for Constants

\[ \text{false} : \text{Bool} \]

\[ s \text{ is a string literal} \]

\[ s : \text{String} \]
More Rules

\[ \vdash e : \text{Bool} \]
\[ \vdash \neg e : \text{Bool} \]

\[ \vdash e_1 : \text{Bool} \]
\[ \vdash e_2 : \text{T} \]
\[ \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool : Object} \]
More Rules

\[
\frac{\Gamma \vdash e : \text{Bool}}{
\Gamma \vdash \neg e : \text{Bool}
}\]

\[
\frac{
\begin{array}{l}
\Gamma \vdash e_1 : \text{Bool} \\
\Gamma \vdash e_2 : \top
\end{array}
}{
\Gamma \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{Object}
}\]

loop returns \text{void}
New

\[ \text{new } T : T \]

(Ignore \texttt{SELF\_TYPE} for now)
Variables

\[ x \text{ is a variable} \]

\[ \vdash x : ? \]

\[ \text{we need to look up the declaration, somehow.} \]

Solution: Add type environment (symbol table) to the rules.
$O$ maps object IDs to types.

If $x$ is a variable, $O(x)$ is its type.

$O \vdash e : T$

"$e$ has type $T$ assuming the types of variables given in $O$."
We can add 0 to previous rules (rules just ignore it)

E.g. \[ 0 \vdash e_1 : \text{Int} \quad 0 \vdash e_2 : \text{Int} \]

\[ 0 \vdash e_1 + e_2 : \text{Int} \]
Variable Rule

\[
\frac{0(x) = T}{0H \vdash x : T}
\]
Let (without initialization)

Notation: \( O[T/y] \) is the function \( O \), modified to return \( T \) on argument \( y \)

\[
O[T/x] \vdash e_1 : T
\]

\[
O \vdash \text{let } x : T_0 \text{ in } e_1 : T_1
\]

Type of \( e_1 \) is computed with \( O[T/x] \). Scope is limited to \( e_1 \).
Let with initialization

\[ O \vdash e_0 : T_0 \]
\[ O[T_0/x] \vdash e_i : T_1 \]
\[ O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_i : T_1 \]

Unnecessarily restrictive

It does not allow \( e_0 \) to be a class that inherits from \( T_0 \).
Subtyping

Partial order for inheritance

\[ T \leq T \text{ (reflexive)} \]
\[ T \leq T' \text{ if } T \text{ inherits from } T' \]
\[ T \leq T'' \text{ if } T \leq T' \text{ and } T' \leq T'' \text{ (transitive)} \]
$$O \vdash e_0 : T_0$$
$$O[T/x] \vdash e_1 : T_1$$
$$T_0 \leq T$$

$$O \vdash \text{let } x : T \leftarrow e_0 \text{ in } e_1 : T_1$$

Allows $e_0$ to have any sub-type of declared type of $x$. 
Assignment

\[ O(x) = T_0 \]
\[ O \vdash e_1 : T_1 \]
\[ T_1 \leq T_0 \]
\[ O \vdash x \Leftarrow e_1 : T_1 \]

\[ \downarrow \]

Allows \( e_1 \) to be any subtype of declared type of \( x \)
Assignment

\[ O(x) = T_0 \]
\[ O \vdash e_i : T_1 \]
\[ T_1 \leq T_0 \]
\[ O \vdash x \leftarrow e_i : T_1 \]

\[ \text{Assignment returns value of } e_i \text{ which has type } T_i. \]
If-then-else

\[ \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \]

At compile-time, don't know which of \( e_1, e_2 \) will be returned.

\( e_1 : T_1 \), \( e_2 : T_2 \) — need a type that is \( T_1 \) OR \( T_2 \)
Least Upper Bound

\[ \text{lub} \left( T_1, T_2 \right) = \text{least type } T_3 \text{ such that } T_1 \leq T_3 \text{ and } T_2 \leq T_3 \]

least \: \forall T_4 \left( T_1 \leq T_4 \: \land \: T_2 \leq T_4 \right) \rightarrow T_3 \leq T_4

Notation \: T_1 \cup T_2 \: ("join" \: of \: T_1, \: T_2)
0 \vdash e_0 : \text{Bool}
0 \vdash e_1 : T_1
0 \vdash e_2 : T_2
\overline{0 \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : T_1 \cup T_2}
Next

Need more stuff in environment

O – already have

M – methods

\[ M(C, f) = (T_1, T_2, \ldots, T_n, T_{n+1}) \]

\[ \uparrow \quad e \]

class method name

\[ \quad \text{types of formals return type} \]

C – class of "self"
Implementation

Environment is passed down AST
Argument to recursive function

Types of expressions are computed
bottom-up.
\[ +c(\text{env}, e_1 + e_2) : \]
\[ T_1 = +c(\text{env}, e_1)_j \]
\[ T_2 = +c(\text{env}, e_2)_j \]
\[ \text{check } T_1 == T_2 == \text{Int} \]
\[ \text{return Int}_j \]