Introduction to Semantic Analysis

The Compiler so far

- Lexical Analysis
  - Detects inputs with illegal tokens
- Parsing
  - Detects inputs with ill-formed parse trees
- Next: Semantic Analysis
  - Last ‘front-end’ phase
  - Catches all the remaining errors

Semantic Analysis

The compilation process is driven by the syntactic structure of the program as discovered by the parser

Semantic routines:

- interpret (partial) meaning of the program based on its syntactic structure
- two purposes:
  - finish analysis by deriving context-sensitive information (e.g. type checking)
  - begin synthesis by generating the IR or target code

What does Semantic Analysis do?

- Checks of many kinds:
  - All identifiers are declared.
  - Type checking in assignment statements, function calls, expressions, etc.
  - Inheritance relationships.
  - Classes defined only once.
  - Methods in a class defined only once.
  - Reserved identifiers are not misused

  And many others...

Why a Separate Semantic Analysis?

Parsing cannot catch some errors. Some language constructs are not context-free.

- All identifiers are declared before use.
  - Corresponds to the abstract language $\{wcw \mid w, c \in \{a, b\}^*\}$.
- The number of formal parameters in the declaration of a function agrees with the number of actual parameters in the use of the function.
  - Corresponds to the abstract language $\{a^n b^m c^n d^m\}$.
  - Here, a^n, b^m represent the formal-parameter list of two functions, while c^n, d^m represent the actual-parameter list in calls to the functions.

These languages are not context-free.

Scope

- Matching identifier declarations with uses
  - One of the most important tasks performed by Semantic Analysis.
- 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 identifer may have restricted scope even if declared before use.

Static vs. Dynamic Scope

- Most languages have static scope.
  - Scope depends only on the program text, not run-time behavior.
  - C,C++,Java, etc. have static scope.

Example:

```
int x = 0;
{
    y = x;
    int x = 1;
    z = x;
}
```

Uses of `x` refer to closest enclosing definition.

Static vs. Dynamic Scope

- Some languages have dynamic scope.
  - Scope depends on the execution of the program.
- A dynamically scoped variable refers to the closest enclosing binding in the execution of the program.
  - Lisp, SNOBOL have dynamic scoping for variables.
- The exception handler for an exception is dynamically determined.
- Object-oriented languages have limited dynamic scoping for identifying method bindings.

Example of a language with dynamic scoping for variables:

```
a ← 0
g(y) = let a ← 4 in f();
h(z) = let a ← 5 in f();
f() = a;
```

Symbol tables

For compile-time efficiency, compilers use a symbol table:

- associates lexical names (symbols) with their attributes

What items belong to the symbol table?

- variable names
- defined constants
- procedure and function names
- source text labels
- compiler-generated temporaries (we'll get there)

A symbol table is a compile-time structure

Symbol table information

What kind of information might the compiler need?

- textual name
- data type
- declaring procedure
- lexical level of declaration
- storage class (base address)
- offset in storage
- can it be aliased? to what other names?
- number and type of arguments to functions
- …

Storage classes of variables

Variables in the program may be assigned different storage classes based on their lifetime, scope, where they live, etc.

Common storage classes are: global, stack, static, registers.

The storage classes also determine how the variables are addressed (addressing mode).

- A local variable is not assigned a fixed machine address (or relative to the base of a module)
- Rather a stack location that is accessed by an offset from a register whose value may be different every time the procedure is invoked.

Symbol table organization

How should the table be organized?

- Linear List
  - **O**(n) probes per lookup
  - easy to expand — no fixed size
  - one allocation per insertion
- Ordered Linear List
  - **O**$(\log_2 n)$ probes per lookup using binary search
  - insertion is expensive (to reorganize list)
- Binary Tree
  - **O**(n) probes per lookup — unbalanced
  - **O**$(\log_2 n)$ probes per lookup — balanced
  - easy to expand — no fixed size
  - one allocation per insertion
- Hash Table
  - **O**(1) probes per lookup — on average
  - expansion costs vary with specific scheme

Nested scopes: block-structured symbol tables

What information is needed?

- when asking about a name, want most recent declaration
- declaration may be from current scope or outer scope
- innermost scope overrides outer scope declarations

Key point: new declarations occur only in current scope

Nested scopes: block-structured symbol tables

What operations do we need?

- `void put (Symbol key, Object value)`
  bind key to value
- `Object get(Symbol key)`
  return value bound to key
- `void beginScope()`
  remember current state of table
- `void endScope()`
  close current scope and restore table to state at most recent open beginScope

Attribute information

Attributes are internal representation of declarations

Symbol table associates names with attributes

Names may have different attributes depending on their meaning:

- variables: type, procedure level, frame offset
- types: type descriptor, data size/alignment
- constants: type, value
- procedures: formals (names/types), result type, block information (local decls.), frame size

Types

- What is a type?
  - A set of values.
  - A set of operations on those values.
- Classes (e.g. in Java) are one instantiation of the modern notion of type.

Type system

Type systems is a logical system comprising of a set of rules that assigns a property called a type to every term in the language.

Why do we need Type Systems?

- Not all operations are legal for all types.
- It makes sense to add two integers, but it doesn't make any sense to add a function pointer and an integer in C.
- But both have the same assembly language implementation: `add $r1, $r2, $r3`.

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 correct types.

Type Checking Overview

- Three kinds of languages:
  - *Statically typed*: All or almost all checking of types is done as part of compilation (C, Java, OCaml, Rust, MiniJava).
  - *Dynamically typed*: Almost all checking of types is done as part of program execution (Python,JavaScript,Scheme)
  - *Untyped*: No type checking (machine code)

Static VS Dynamic Type Checking

- Static checking catches many programming errors at compile time.
- Static checking also avoids overhead of runtime type checks.
- Dynamic type systems are more flexible than static type systems – allow more programs to be executed.
- Dynamically typed languages allow rapid prototyping, but may be difficult to develop and maintain large code bases.
- Recent trend: bolt-on static type checking on top of dynamically typed languages.
  - See Hack (Meta, PHP), Flow (Meta, JavaScript), TypeScript (Microsoft, JavaScript), Eqwalizer (WhatsApp, Erlang), Pyre (Meta, Python).

This is a type war. There is no clear winner.

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 (incorrectly) used interchangeably.

- The formalism to express both type-checking and type-inference is *logical rules of inference*.
  - Just like how the formal notation for lexical analysis and parsing were regular expressions and context-free grammars respectively.

Rules of Inference

- Inference rules have the form:
  If Hypothesis is true, then Conclusion is true.
  - Hypothesis is also called as *antecedent*
  - Conclusion is also called as *consequent*

- We use the notation $\vdash e : T$ to denote that e has type T, and the notation $\vdash e$ to denote that e type-checks.

- Inference rules are often written as follows:

$$\frac{\vdash \text{Hypothesis}_1 \quad \vdash \text{Hypothesis}_2 \quad \ldots \quad \vdash \text{Hypothesis}_n}{\vdash \text{Conclusion}}$$

This translates to: **If** Hypothesis_1 **and** Hypothesis_2 **and** $\ldots \text{Hypothesis}_n$ **then** Conclusion.

Inference rules are a very succinct way to precisely state type-checking rules.

Rules of Inference

$$\frac{\vdash E1 : T1 \quad \vdash E2 : T2}{\vdash E3 : T3}$$

- For type inference, the rule is read as:
  - *If* $E1$ *has type* $T1$ *and* $E2$ *has type* $T2$ *then* $E3$ *has type* $T3$*.*
- For type checking, the rule is read as:
  - *If* E_1 *with type* $T1$ *and* E_2 *with type* $T2$ *type-check, then* E_3 *also type-checks with type* $T3$*.*
  - We may sometimes omit $T3$ when type checking (statements, for example).

Examples

$$\frac{\text{program contains } \texttt{int i}}{\vdash \texttt{i} : int}$$

$$\frac{\vdash e_1 : int \quad \vdash e_2 : int}{\vdash e_1 + e_2 : int}$$

$$\frac{\vdash id : int \quad \vdash e : int}{\vdash id = e}$$

Type Environment

- A type environment is a mapping from identifiers to types.
  - We will use the symbol A to denote type environment.
  - $dom(A)$ denotes the domain of A, i.e. the set of identifiers for which types are defined in the environment A.
  - We use the notation $A \vdash id : T$ to denote that $A(id) = T$.

Type expressions

Type expressions are a textual representation for types:

1. basic types: $boolean$, $char$, $integer$, $real$, etc.
2. type names
3. constructed types (constructors applied to type expressions):
   1. $array(I, T)$ denotes an array of T indexed over I
      e.g., $array(1 \ldots 10, integer)$
   2. products: $T_1 \times T_2$ denotes Cartesian product of type expressions T_1 and T_2
   3. records: fields have names
      e.g., $record((\texttt{a} \times integer), (\texttt{b} \times real))$
   4. pointers: $pointer(T)$ denotes the type "pointer to an object of type T"
   5. functions: $D \rightarrow R$ denotes the type of a function mapping domain type D to range type R
      e.g., $integer \times integer \rightarrow integer$

In Minijava, the type expressions are `boolean`, `int`, `int[]`, id.

MiniJava Type Judgements

	$\vdash$	g	The goal g type checks
	$\vdash$	mc	The main class mc type checks
	$\vdash$	d	The type declaration d type checks
C	$\vdash$	m	If defined under class C, the method m type checks
A, C	$\vdash$	s	In a type environment A, if written in class C, the statement s type checks
A, C	$\vdash$	$e : t$	In a type environment A, if written in class C, the expression e has type t
A, C	$\vdash$	$p : t$	In a type environment A, if written in class C, the primary expression e has type t

Typechecking a Minijava program

A program in Minijava consists of a main class (which contains the `main` method) and a bunch of other classes.

Our goal is to typecheck all the classes:

$$\frac{distinct(classname(mc), classname(d_1), \ldots, classname(d_n)) \quad \vdash mc \quad \vdash d_i, i \in \{1, \ldots, n\}}{\vdash mc\ d_1 \ldots d_n}$$

This says that if all the class-names are distinct, and if all the classes individually type-check, then the entire program also type-checks.

Ref: The MiniJava Type System, Jens Palsberg.

Typechecking a Minijava class

$$\frac{distinct(id_1, \ldots, id_f) \quad distinct(methodname(m_1), \ldots, methodname(m_k)) \quad id \vdash m_i \quad i \in \{1, \ldots, k\}}{\vdash \texttt{class}\ \ id\{t_1\ id_1;\ \ldots;\ t_f\ id_f;\ m_1\ \ldots m_k\}}$$

$id \vdash m_i$ indicates that method m_i type-checks under a type-environment consisting of fields of class id.

Question: Does this allow a method and a field to have the same name?

Ref: The MiniJava Type System, Jens Palsberg.

“fields” helper function

$$\frac{\texttt{class}\ id\ \{\ t_1\ id_1\texttt{;}\ \ldots\texttt{;}\ t_f\ id_f\texttt{;}\ m_1\ \ldots\ m_k\ \}\quad \text{is in the program}}{fields(id) = [id_1 : t_1, \ldots, id_f : t_f]}$$

$$\frac{\texttt{class}\ id\ \texttt{extends}\ id^P\ \{\ t_1\ id_1\texttt{;}\ \ldots\texttt{;}\ t_f\ id_f\texttt{;}\ m_1\ \ldots\ m_k\ \}\quad \text{is in the program}}{fields(id) = fields(id^P) \cdot [id_1 : t_1, \ldots, id_f : t_f]}$$

Typechecking a Minijava method

$$\frac{\begin{array}{c} distinct(id_1^F, \ldots, id_n^F) \\ distinct(id_1, \ldots, id_r) \\ A = fields(C) \cdot [id_1^F : t_1^F, \ldots id_n^F : t_n^F] \cdot [id_1 : t_1, \ldots id_r : t_r] \\ A, C \vdash s_i,\ i \in \{1, \ldots, q\} \quad A, C \vdash e : t \end{array}}{\begin{array}{l} C \vdash \texttt{public}\ \ t\ id^M\ (t_1^F\ id_1^F,\ \ldots,\ t_n^F\ id_n^F)\{ \\ t_1\ id_1;\ \ldots;\ t_r\ id_r;\ s_1\ \ldots s_q;\ \texttt{return}\ \ e;\} \end{array}}$$

Question:

- Does this allow a formal parameter and a local variable to have the same name?
  - What will be its type?
- Does this allow a local variable and a field to have the same name?

Ref: The MiniJava Type System, Jens Palsberg.

Inheritance in Minijava

To express inheritance among classes, we will define the *subtype* relation, denoted by $\leq$.

$t_1 \leq t_2$ iff t_1 is a sub-class of t_2.

The following rules describe the *subtype* relation:

$$t \leq t$$

$$\frac{t_1 \leq t_2 \quad t_2 \leq t_3}{t_1 \leq t_3}$$

$$\frac{\texttt{class } C \texttt{ extends } D \quad \text{is in the program}}{C \leq D}$$

Typechecking statements in Minijava

$$\frac{A(id) = t_1 \qquad A, C \vdash e : t_2 \qquad t_2 \leq t_1}{A, C \vdash id \texttt{ = } e\texttt{;}}$$

$$\frac{A(id) = \texttt{int[]} \qquad A, C \vdash e_1 : \texttt{int} \qquad A, C \vdash e_2 : \texttt{int}}{A, C \vdash id \texttt{ [} e_1 \texttt{] = } e_2\texttt{;}}$$

$$\frac{A, C \vdash e : \texttt{boolean} \qquad A, C \vdash s_1 \qquad A, C \vdash s_2}{A, C \vdash \texttt{if (} e \texttt{) } s_1 \texttt{ else } s_2}$$

$$\frac{A, C \vdash e : \texttt{boolean} \qquad A, C \vdash s}{A, C \vdash \texttt{while (} e \texttt{) } s}$$

$$\frac{A, C \vdash e : \texttt{int}}{A, C \vdash \texttt{System.out.println(} e \texttt{);}}$$

Ref: The MiniJava Type System, Jens Palsberg.

Typechecking expressions in Minijava

$$\frac{A, C \vdash p_1 : \texttt{boolean} \qquad A, C \vdash p_2 : \texttt{boolean}}{A, C \vdash p_1 \ \texttt{\&\&} \ p_2 : \texttt{boolean}}$$

$$\frac{A, C \vdash p_1 : \texttt{int} \qquad A, C \vdash p_2 : \texttt{int}}{A, C \vdash p_1 \ \texttt{<} \ p_2 : \texttt{boolean}}$$

$$\frac{A, C \vdash p_1 : \texttt{int} \qquad A, C \vdash p_2 : \texttt{int}}{A, C \vdash p_1 \ \texttt{+} \ p_2 : \texttt{int}}$$

$$\frac{A, C \vdash p_1 : \texttt{int} \qquad A, C \vdash p_2 : \texttt{int}}{A, C \vdash p_1 \ \texttt{-} \ p_2 : \texttt{int}}$$

$$\frac{A, C \vdash p_1 : \texttt{int} \qquad A, C \vdash p_2 : \texttt{int}}{A, C \vdash p_1 \ \texttt{*} \ p_2 : \texttt{int}}$$

$$\frac{A, C \vdash p_1 : \texttt{int[]} \qquad A, C \vdash p_2 : \texttt{int}}{A, C \vdash p_1 \ \texttt{[} \ p_2 \ \texttt{]} : \texttt{int}}$$

$$\frac{A, C \vdash p : \texttt{int[]}}{A, C \vdash p \ \texttt{.length} : \texttt{int}}$$

Typechecking method calls in Minijava

$$\frac{???}{A, C \vdash p.id(e_1, e_2, \ldots, e_n) : t}$$

Typechecking method calls in Minijava

$$\frac{A, C \vdash p : D \quad methodtype(D, id) = (t_1', \ldots, t_n') \to t \qquad A, C \vdash e_i : t_i \quad t_i \leq t_i', \ i \in \{1, \ldots, n\}}{A, C \vdash p.id(e_1, e_2, \ldots, e_n) : t}$$

$methodtype(D, id)$ returns the type of the method id in class D or one of its super-classes.

Question: What change should we make in the rule if we want to enforce access modifiers?

Typechecking method calls in Minijava with access modifiers

$$\frac{\begin{array}{c} A, C \vdash p : D \quad methodtype(D, id) = (t_1', \ldots, t_n') \to t \\ (methodaccess(D, id) = \texttt{public} \\ \vee\ methodaccess(D, id) = \texttt{protected} \wedge C \leq D \\ \vee\ methodaccess(D, id) = \texttt{private} \wedge C = D) \\ A, C \vdash e_i : t_i \quad t_i \leq t_i' \ i \in \{1, \ldots, n\} \end{array}}{A, C \vdash p.id\ (e_1, e_2, \ldots, e_n) : t}$$

$methodtype(D, id)$ returns the type of the method id in class D or one of its super-classes.

$methodaccess(D, id)$ returns the access modifier of the method id in class D or one of its super-classes.

Typechecking ternary operation in Minijava

$$\frac{???}{A, C \vdash (e_1 ? e_2 : e_3) : t}$$

Typechecking ternary operation in Minijava

$$\frac{A, C \vdash e_1 : \texttt{boolean} \quad A, C \vdash e_2 : t \quad A, C \vdash e_3 : t}{A, C \vdash (e_1 ? e_2 : e_3) : t}$$

But what about inheritance? How to handle subtyping?

Example

```
...
class B extends A {...}
class C extends B {...}
class D extends B {...}
...
A objA; B objB; C objC; D objD;
...
objA = (...) ? objC : objD;
objB = (...) ? objC : objD;
```

Both the instances of ternary statement should type-check. This requires the ternary statement itself to have the type `B`.

Least Upper Bounds

- $lub(C, D)$, the least upper bound of classes C and D is defined to class E such that
  1. $C \leq E$ and $D \leq E$: E is an upper bound of both C and D
  2. $\forall E'.\ C \leq E' \wedge D \leq E' \Rightarrow E \leq E'$: E is least among upper bounds.
- In Minijava, the least upper bound of two classes is their least common ancestor in the inheritance tree.

Typechecking ternary operation in Minijava: Attempt-2

$$\frac{A, C \vdash e_1 : \texttt{boolean} \quad A, C \vdash e_2 : t_1 \quad A, C \vdash e_3 : t_2 \qquad \exists t' .\ t_1 \leq t' \wedge t_2 \leq t' \quad t = lub(t_1, t_2)}{A, C \vdash (e_1 ? e_2 : e_3) : t}$$

Typechecking No-overloading in Minijava

- **Overloading:** A method is said to be overloaded if there exists another method in the same class or its super classes with the same name but different type.
- **Overriding:** A method in a class is said to be overridden if there exists a method in its subclass with the same name and type.

$$\frac{\begin{array}{c} distinct(id_1, \ldots, id_f) \\ distinct(methodname(m_1), \ldots, methodname(m_k)) \\ id \vdash m_i,\ i \in \{1, \ldots, k\} \\ ??? \end{array}}{\vdash \texttt{class}\ id\ \texttt{extends}\ id^P\{t_1\ id_1;\ \ldots;\ t_f\ id_f;\ m_1\ \ldots m_k\}}$$

Typechecking No-overloading in Minijava

$$\frac{\begin{array}{c} distinct(id_1,\ldots,id_f) \\ distinct(methodname(m_1),\ldots,methodname(m_k)) \\ id \vdash m_i,\ i \in \{1,\ldots,k\} \\ methodtype(m_i,id^P) \neq \bot \Rightarrow methodtype(m_i,id^P) = methodtype(m_i,id) \\ \wedge\ methodaccess(m_i,id^P) = methodaccess(m_i,id),\ i \in \{1,\ldots,k\} \end{array}}{\vdash \texttt{class}\ id\ \texttt{extends}\ id^P\{t_1\ id_1;\ \ldots;\ t_f\ id_f;\ m_1\ \ldots m_k\}}$$

$methodtype(D,id)$ returns

- the type of the method id in class D or one of its super-classes.
- $\bot$ if the method id is not defined in class D or one of its super-classes.

Soundness of a Type System

- A type system is *sound* if whenever $\vdash e : T$, then e will always evaluate to a value of type T in all executions.
- We want only sound rules, because then we can give a compile-time guarantee for the absence of *type errors*.

- Example of an unsound rule?

$$\frac{\vdash e_1 : int \quad \vdash e_2 : int}{\vdash e_1 + e_2 : int[]}$$

- A programming language is called *type safe* if a compiler of the language guarantees that compiled programs will run without type errors.
  - Can be either statically or dynamically typed.