Semantic Analysis

Lecture 13
(no lecture 12)

Dr. Sean Peisert – ECS 142 – Spring 2009
Status

• Project 2 was last weekend... grades in a week or so
• Regrades for project 1 in process.
• Midterm on Friday, May 1 in class (be on time!!)
• Project 3 Due Friday, May 22, 11:55pm
• Read Ch. 5 by next week.
• My office hours this afternoon at 3pm.
• Discussion section this afternoon at 4:10pm.
What is semantic checking?

- Syntax checking (parsing) makes sure that the code fits a given grammatical structure.
- Semantic checking makes certain that the words used in a given piece of code make sense when used together.
The Compiler So Far

- Lexical analysis
  - Detects inputs with illegal tokens
- Parsing (syntax analysis)
  - Detects inputs with ill-formed parse trees
- Semantic analysis
  - Last “front end” phase
  - Catches more errors
What’s Wrong?

• Example 1
  let y: Int in x + 3

• Example 2
  let y: String ← “abc” in y + 3
Why a separate semantic analysis?

• Parsing cannot catch some errors
• Some language constructs are not context-free (in fact, they are context-dependent)
  • Example: all used variables must have been declared (i.e., scoping)
  • Example: a method must be invoked with arguments of proper type (i.e., typing)
What does semantic analysis do?

- Checks of many kinds... **coolc** checks:
  1. All identifiers are declared
  2. Types
  3. Inheritance relationships
  4. Classes defined only once
  5. Methods in a class defined only once
  6. Reserved identifiers are not misused
  - ...and others

- The requirements depend on the language
Scope

- Matching identifier declarations with uses
- Important semantic analysis step in most languages
- Including COOL!
• The scope of an identifier is the portion of a program in which an identifier is accessible.

• The same identifier may refer to different things in different parts of the program.

• An identifier may have restricted scope
Static vs. Dynamic Scope

- Most languages have static scope.
  - Scope depends on the program text, not run-time behavior
  - Cool has static scope
- A few languages are dynamically scoped
  - Scope depends on execution of the program.
Static Scoping Example

```plaintext
let x: Int ← 0 in
{
    x;
    let x: Int ← 1 in
    x;
    x;
    x;
}
```
Static Scoping Example

```swift
let x: Int ← 0 in
{
    x;
    let x: Int ← 1 in
        x;
        x;
        x;
    }
}

• Uses of x refer to closest enclosing definition.
Dynamic Scope

• A dynamically-scoped variable refers to the closest enclosed binding in the execution of the program.

• Example:

\[ g() = \text{let } a \leftarrow 4 \text{ in } f(); \]
\[ f() = a + 2; \]

When invoking \( g() \) the result will be 6.

• More about dynamic scope later in the course
Scope in Cool

- Cool identifier bindings are introduced by:
  - Class declarations (introduce class names)
  - Method definitions (introduce method names)
  - Let expressions (introduce object ids)
  - Formal parameters (introduce object ids)
  - Attribute definitions in a class (introduce object ids)
  - Case expressions (introduce object ids)
Implementing the Most-Closely Nested Rule

• Much of semantic analysis can be expressed as a recursive descent of an AST
  • Process an AST node $n$
  • Process the children of $n$
  • Finish processing the AST node $n$
Implementing

• Example: the scope of “let” bindings is one subtree:

  let x : Int ← 0 in e

• x can be used in subtree e
Symbol Tables
Symbol Table

- A symbol table handles creating and merging scopes (e.g., the parent's scope with the child's)
- When an item is added to the symbol table, it will handle inserting the item into the current scope.
- When you look up an identifier in the symbol table to see if it is visible, it will handle looking in the current scope and the parent's scope.
Symbol Table

- The Symbol Table is an external data structure that maintains information about the identifiers in a program.
- It should not be part of the parser, because the symbol table is used in all areas of the compiler.
- It is a database, of sorts.
Symbol Table

- Consider again: let x : Int ← 0 in e

- Idea:
  - Before processing e, add definition of x to current definitions, overriding any other definition of x.

- A symbol table is a data structure that tracks the current bindings of identifiers

- We will give you an implementation for the project.
Scope in Cool

• Not all kinds of identifiers follow the most-closely nested rule

• For example, class definitions in Cool:
  • cannot be nested
  • are globally visible throughout the program

• In other words, a class name can be used before it is defined
Example: Use Before Definition
(this is legal in Cool)

Class A {
    ... let y : B in ...
};

Class B {
    ...
};
More Scope in Cool

- Attribute names are global within the class in which they are defined.

```java
Class X {
    f(): Int {a};
    a: Int ← 0;
}
```
More Scope

• Method and attribute names have complex rules.

• A method need not be defined in the class in which it is used, but in some parent class.

• Methods may also be redefined (overridden).
Class Definitions

- Class names can be used before being defined
- We can’t check this property
  - using a symbol table
  - or even in one pass
- Solution
  - Pass 1: Gather all class names
  - Pass 2: Do the checking
- Semantic analysis requires multiple passes
  - Probably more than two
Types

• What is a type?
  • The notion varies from language to language
• Consensus
  • A set of values
  • A set of operations on these values
• Classes are one instantiation of the modern notation of type.
Why do we need type systems?

• Consider the assembly language fragment:

\[
\text{addi } r1, r2, r3
\]

• What are the types of $r1, r2, and $r3
Types and Operations

• 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!

• Type systems provide a concise formalization of semantic checking rules.
What can types do for us?

- Can detect certain kinds of errors
  - Memory errors:
    - Reading from an invalid pointer, etc..
  - Violation of abstraction boundaries:

```java
class FileSystem {
    open(x: String): File {
        ...
    }
    ...
}

class Client {
    f(fs: FileSystem) {
        File fdesc ← fs.open("myfile")
        ...
        } -- f cannot see inside fdesc
    }
```
Type Checking
Overview

• Three kinds of languages:
  
  • *Statically typed*: all or almost all checking of types is done as part of compilation (C, Java, Cool)
  
  • *Dynamically typed*: almost all checking of types is done as part of program execution (Scheme)
  
  • *Untyped*: No type checking (machine code)
Type Wars

- Competing views on 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 easier in a dynamic type system
Type Wars

- In practice, most code is written in statically-typed languages with an “escape” mechanism
  - Unsafe casts in C
  - Native methods in Java
  - pyobjc for C / Python
Cool Types

• The types are:
  • Class names
  • SELF_TYPE
  • Note: there are no base types (as in Java int, ...)
• The user declares types for all identifiers
• The compiler infers types for expressions
• Infers a type for every expression.
Type Checking and Type Inference

- *Type checking* is the process of verifying fully typed programs
- *Type inference* is the process of filing in missing type information
- The two are different, but are often used interchangeably.
Rules of Inference

• We have seen two examples of formal notation for specifying parts of a compiler.
  • Regular expressions (for the lexer)
  • Context-free grammars (for the parser)

• The appropriate formalism for type checking is logical rules of inference.
Why rules of inference?

- Inference rules have the form
  - *If hypothesis is true, the conclusion is true.*

- Type checking computes via reasoning:
  - *If $E_1$ and $E_2$ have certain types, then $E_3$ 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 $\land$ is “and”
  - Symbol $\Rightarrow$ is “if-then”
  - $x:T$ is “$x$ has type $T$”
From English to an Inference Rule

If $e_1$ has type Int and $e_2$ has type Int, then $e_1 + e_2$ has type Int

$$(e_1 \text{ has type } \text{Int} \land e_2 \text{ has type } \text{Int}) \implies e_1 + e_2 \text{ has type } \text{Int}$$

$$(e_1 : \text{Int} \land e_2 : \text{Int}) \implies (e_1 + e_2) : \text{Int}$$
From English to an Inference Rule

- This statement

  \[(e_1: \text{Int} \land e_2: \text{Int}) \Rightarrow (e_1 + e_2) : \text{Int}\]

- is a special case of

  \[(\text{Hypothesis}_1: \text{Int} \land \ldots \text{Hypothesis}_n) \Rightarrow \text{Conclusion}\]

- This is an inference rule.
Notation for Inference Rules

• By tradition, inference rules are written

  “if” \[ \vdash \text{Hypothesis}_1 \quad \ldots \quad \vdash \text{Hypothesis}_n \]  

  “then” \[ \vdash \text{Conclusion} \]

• Cool type rules have hypotheses and conclusions of the form:

  \[ \vdash e : T \]

• means “it is provable that ...”
Two Rules

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

\[ \vdash e_1 : \text{Int} \]
\[ \vdash e_2 : \text{Int} \]
\[ \vdash e_1 + e_2 : \text{Int} \]
Two Rules

• These rules give templates describing how to type integers and + expressions

• By filling in the templates, we can produce complete typings for expressions.
Example: \(1 + 2\)

\[
\begin{align*}
\text{\small 1 is an integer} & \quad \text{\small 2 is an integer} \\
\quad \quad \text{\small \(\vdash 1: \text{Int}\)} & \quad \quad \text{\small \(\vdash 2: \text{Int}\)} \\
\quad \quad \quad \text{\small \(\vdash 1 + 2: \text{Int}\)}
\end{align*}
\]
Soundness

- A type system is **sound** if
  - Whenever $\vdash e : T$
  - Then $e$ evaluates to a value of type $T$
- We only want sound rules
  - But some sound rules are better than others:
    - $i$ is an integer
      $\vdash i : \text{Object}$
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 kind of AST node
- In the type rule used for a node $e$:
  - The hypotheses are the proofs of types of $e$’s subexpressions
  - The conclusion is the proof of type of $e$
- Types are computed in a bottom-up pass over the AST
Rules for Constants

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

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

\[ \vdash s : \text{String} \]
Rule for New

- new T produces an object of type T
- Ignore SELF_TYPE for now...

\[ \vdash \text{new } T : T \]
Two More Rules

\[ \begin{align*}
\Gamma &\vdash e : \text{Bool} \\
\Gamma &\vdash \text{not } e : \text{Bool} \\
\Gamma &\vdash e_1 : \text{Bool} \\
\Gamma &\vdash e_2 : T \\
\Gamma &\vdash \text{while } e_1 \text{ loop } e_2 \text{ pool : Object} \\
\end{align*} \]

[Not]

[Loop]
Tackling this Project

- This is a long project and there are a lot of details.
- There are ~4 weeks in which to finish the project. Set deadlines for yourself. Start now.
Timeframe for the rest of the quarter

- Midterm on May 1
- Project 4 due May 22
- No class May 25 (Memorial Day)
- Project 5 due June 5
- No class June 5
- Probably a final exam review sometime
- Final Exam on June 11