

Overview of Semantic Analysis

Lecture 9

Midterm Thursday

- In class
 - SCPD students come to campus for the exam
- Material through lecture 8
- Open note
 - Laptops OK, but no internet or computation

Outline

- The role of semantic analysis in a compiler
 - A laundry list of tasks
- Scope
 - Implementation: symbol tables
- Types

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 - last line of defense
- So in a sense, these are filters that reject input strings, leaving only valid programs

Why a Separate Semantic Analysis?

- Parsing cannot catch some errors
- Some language constructs not context-free
- Situation is very similar to moving from lexical analysis to parsing phases: some things cannot be expressed by CFGs, so we need something more to catch these

What Does Semantic Analysis Do?

- Checks of many kinds (this is typical).
- **coolc** checks
 1. All identifiers are declared (and scope restrictions observed)
 2. Type Checking (major functions of semantic analyzer for **cool**)

What Does Semantic Analysis Do?

- **coolc checks:**
 3. Inheritance relationships (this and next two are common issues due to object-oriented nature of language)
 4. Classes defined only once
 5. Methods in a class defined only once
 6. Reserved identifiers are not misused (common issue for most languages)

And others (which we will also be discussing. . .)
- **Main message: semantic analyzers do quite a few checks**
 - The requirements depend on the language

Caveat:

- Checks listed in the previous slides are common for statically typed languages, but other classes of languages will have other kinds of checks

Scope

- Motivating problem for scope is that we want to match identifier **declarations** with **uses** of those identifiers
 - Want to know which variable X we're talking about if variable X might have more than one definition
 - Important static analysis step in most languages
 - Including *COOL*!

What's Wrong?

- Example 1

Let $y: \text{String} \leftarrow \text{"abc"}$ in $y + 3$

Note that declaration will be matched with use, so this will generate an error, since trying to add a string to a number

What's Wrong?

- Example 2

Let $y: \text{Int}$ in $x + 3$

Here's a declaration of y , but no use of y (which is not an error - though we might want to generate a warning).
But there is use of x , with no declaration here.
Where is the definition? If no outer declaration of x , then we should generate an undeclared variable error.

Note: An example property that is not context free.

What's Wrong?

- Example 2

Let $y: \text{Int}$ in $x + 3$

Here's a declaration of y , but no use of y (which is not an error - though we might want to generate a warning).

But there is use of x , with no declaration here.

Where is the definition? If no outer declaration of x , then we should generate an undeclared variable error.

- *Specifically*, whether this statement generates an error depends on whether there is an outer declaration of x . In other words, it depends on context, so it's not context free!

Scope (Cont.)

- 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 can't overlap
 - So a given variable x can only refer to one thing in a given part of the program
- An identifier may have restricted scope
 - Think of examples where scope of variable is less than the entire program

Static vs. Dynamic Scope

- Most languages have *static scope*
 - Scope depends only on the program text, not runtime behavior
 - Cool has static scope
 - Though probably every language you have used to this point is statically scoped, it may come as a surprise that there are alternatives to static scoping...

Static vs. Dynamic Scope

- A few languages are *dynamically* scoped
 - For a while there was some argument over which type of scoping was better. It would seem that static scoping has won that argument.
 - Lisp, SNOBOL
 - Lisp has changed (a long time ago) to mostly static scoping
 - Dynamic scope depends on execution of the program

Static Scoping Example

```
let x: Int <- 0 in
{
  x;
  let x: Int <- 1 in
    x;
  x;
}
```

Three uses of x . Which refer to which definition?

Static Scoping Example (Cont.)

```
let x: Int <- 0 in
{
  x;
  let x: Int <- 1 in
    x;
  x;
}
```

Uses of `x` refer to closest enclosing definition

Most closely nested rule: variable binds to definition (of same name) that is most closely enclosing it. ¹⁷

Dynamic Scope

- A dynamically-scoped variable refers to the closest enclosing binding in the execution of the program (the most recent binding of the variable)
- Example
 - $g(y) = \text{let } a \leftarrow 4 \text{ in } f(3);$
 - $f(x) = a;$
 - Note a defined in some function g . f isn't in same syntactic scope (could be anywhere)
 - Question: what is the value of a when used in f ?
- More about dynamic scope later in the course
 - After we know more about language implementation

Scope in Cool

- Cool identifier bindings are introduced by several mechanisms...
 - Class declarations (introduce class names)
 - Method definitions (introduce method names)
 - Let expressions (introduce object ids)
 - Formal parameters (introduce object ids)
 - Attribute definitions (introduce object ids)
 - Case expressions (introduce object ids)

Scope in Cool (Cont.)

- Not all kinds of identifiers follow the most-closely nested rule
- For example, a rather large exception to this rule is class definitions in Cool
 - Cannot be nested
 - *Are globally visible* throughout the program
 - Available for use anywhere in the program
- In fact, a class name can be used before it is defined

Example: Use Before Definition

```
Class Foo {  
  ... let y: Bar in ...  
};
```

```
Class Bar {  
  ...  
};
```

Bar is used before it is defined, and this is perfectly OK

More Scope in Cool

Similarly, attribute names are global within the class in which they are defined

```
Class Foo {  
  f(): Int { a };  
  a: Int ← 0;  
}
```

method `f` uses `a` before it is defined, and again this is perfectly OK

Normally, we list attribute definitions before method definitions, but that is not required

More Scope (Cont.)

- Method/attribute names have quite complex rules
- Ex. 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)
 - Gives method a new definition, even though it has been defined before
- We don't yet have the language to talk about these rules with any precision (but we will)²³

Shortly...

- We will begin talking about symbol tables.
- But first, we introduce an algorithm that we will use over and over for the rest of the course
 - And that, as it turns out, helps us a bit with scoping rules

Implementing the Most-Closely Nested Rule

- Much of semantic analysis (and a lot of code generation) can be expressed as a **recursive descent** of an AST. At each step, we are processing a node in the AST:

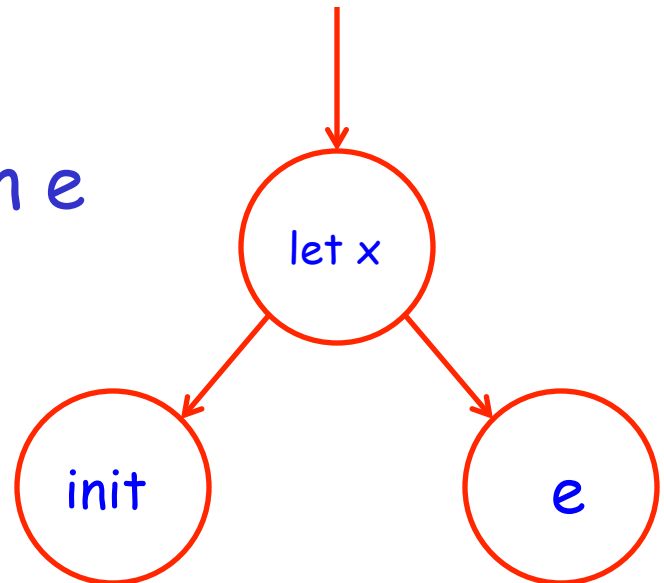
- *Before*: Begin processing an AST node n
- *Recurse*: Process the children of n
- *After*: Finish processing the AST node n

- When performing semantic analysis on a portion of the the AST, we need to know which identifiers are defined

Implementing . . . (Cont.)

- An example of this recursive descent strategy is how we process **let** bindings to track the set of variables that are in scope.
- **let** binding is one subtree of the AST:

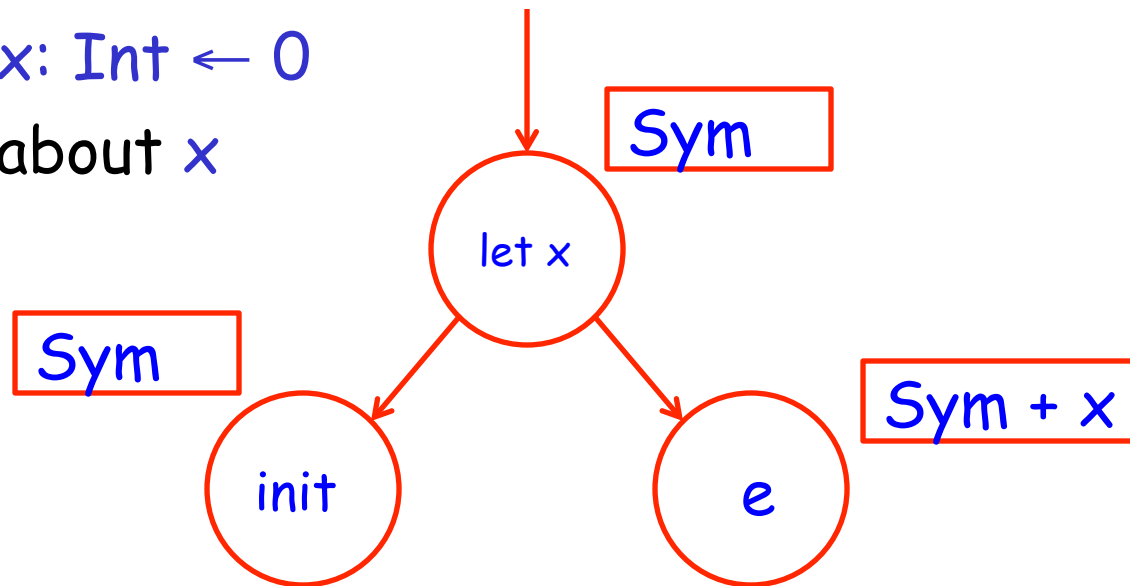
let x : Int \leftarrow 0 in e



- x is defined in subtree e

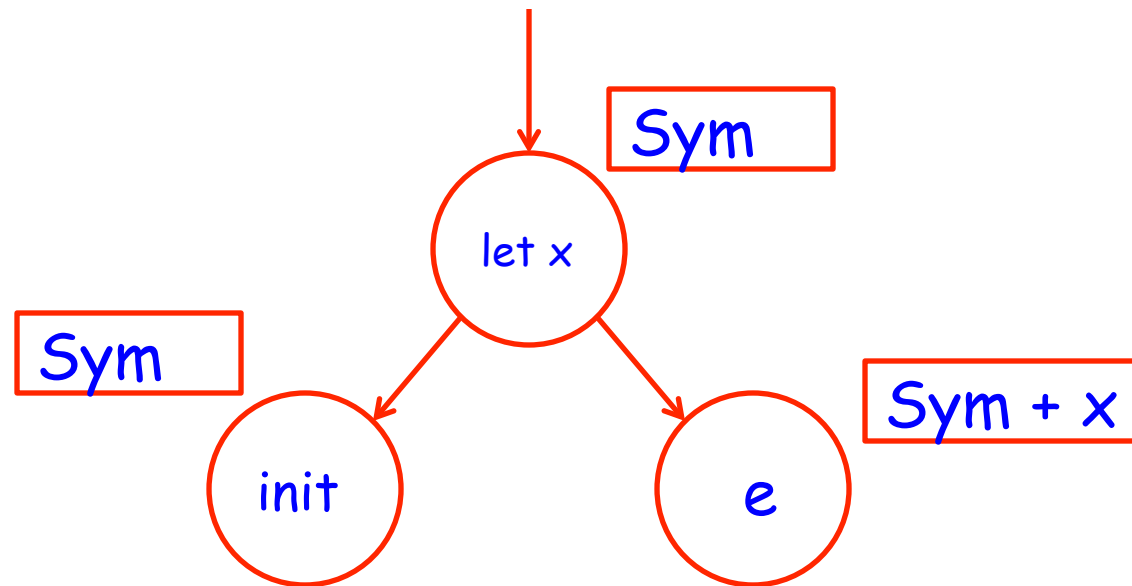
Implementing . . . (Cont.)

- We are processing our AST, when we come to the **let** node
- Some information about symbols (the **symbol table**) is passed to the **let** node
- That info is passed to **init** node
- **init** node processes $x: \text{Int} \leftarrow 0$
- **Sym** table plus info about x passed to **e** node
- When done processing **e** node, **Sym** table returned to original state



Implementing . . . (Cont.)

- So note that when we are done processing this portion of the AST, we leave with the *Sym* table being in exactly the same state it was when we entered this subtree.



Symbol Tables (in terminology of 3 part alg.)

- 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`
 - *Recurse*
 - *After* processing `e`, remove definition of `x` and restore old definition of `x`
- A *symbol table* is a data structure that tracks the current bindings of identifiers

A Simple Symbol Table Implementation

- Structure is 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
 - Note that this takes care of hiding of old definitions
 - `remove_symbol()` pop the stack
 - Leaves stack in same state it was before processing node
- Why does this work?

Limitations

- The simple symbol table works for `let`
 - Because symbols added one at a time
 - Because declarations are perfectly nested
 - This is really the whole reason we can use a stack for `let`
- What doesn't it work for?

Limitations

- The simple symbol table works for `let`
 - Because symbols added one at a time
 - Because declarations are perfectly nested
 - This is really the whole reason we can use a stack for `let`
- What doesn't it work for? Well, consider the following (illegal) piece of code

```
f(x : Int, x: Int) { }
```

- **Detecting** this would be difficult with stack
 - Because functions introduce multiple names at once into the same scope, while stack adds one at a time

Limitations

- The simple symbol table works for `let`
 - Because symbols added one at a time
 - Because declarations are perfectly nested
 - This is really the whole reason we can use a stack for `let`
- What doesn't it work for? Well, consider the following (illegal) piece of code

```
f(x : Int, x: Int) { }
```

- Detecting this would be difficult with stack
 - So both instances would be pushed on stack, one after the other, with no indication that there is an error

A Fancier Symbol Table (Solves the problem)

- `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**
- Biggest change here is explicit enter and exit scope functions.
- New structure is a stack of **scopes**
 - each entry in stack is an entire scope
 - What's in a scope is all the variables that are defined at the same level, within that single scope

A Fancier Symbol Table (Solves the problem)

- `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**
-
- Note that `check_scope()` allows us to detect the kind of errors where `x` is defined twice in the same scope

A Fancier Symbol Table (Solves the problem)

- `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

I supply a symbol table manager for your project (using this same structure, with the interface already provided if you don't want to write your own).

(I'm sure that thought warms you.)

Class Definitions

- Class names behave differently than variables introduced in let bindings and in function parameters
 - Class names can be used before being defined
- As a consequence, we can't check class names
 - using a symbol table
 - or even in one pass (because we can't know if all of the used classes are defined until we've seen the entire file)

Class Definitions

- Solution: Two passes over the program
 - Pass 1: Gather all class names
 - Pass 2: Do the checking
- Lesson: Semantic analysis requires multiple passes
 - Probably more than two
 - So don't be afraid to write a compiler that makes multiple simple passes if this makes your life easier
 - Better than one very complicated pass with entangled code
 - Often much easier to debug your compiler if you're willing to make multiple passes over the input

Let us now begin our introduction to types.

Types

- What is a type?
 - The notion varies from language to language
- Consensus
 - A set of values
 - A set of operations on those values
 - Or perhaps unique to those values
 - Often this issue of operations is important
- Classes are one instantiation of the modern notion of type
 - But this is just one way that modern programming languages express the notion of type

Type Examples

- int
 - Operations include $+$, $-$, $*$, $/$, $>$, $<$, ...
- String
 - Operations include concatenation, test whether string is empty, string length, charAt, indexOf, ...
- Operations on these are different, and we'd like to avoid mixing them up
- In Cool, class names are all the types
 - With the exception of **self-type**

Types (cont.)

- In Cool, class names are all the types
 - With the exception of **self-type**
- Note that this need not be the case for all programming languages
 - Though it's often convenient in OO languages to equate class names with types, this need not be the case
 - There are other designs where class names are not the only types
 - And in other languages, whether there is no notion of class, types are completely different things
 - But be aware that even for OO languages, equating classes and types is not the only possible design

Why Do We Need Type Systems?

Consider the assembly language fragment

add \$r1, \$r2, \$r3

(adds \$r2 to \$r3 and puts result in \$r1)

What are the types of \$r1, \$r2, \$r3?

Why Do We Need Type Systems?

Consider the assembly language fragment

add \$r1, \$r2, \$r3

(adds \$r2 to \$r3 and puts result in \$r1)

What are the types of \$r1, \$r2, \$r3?

Trick question: We might hope they are integers, but at assembly level, we can't tell. Moreover, regardless of what type they are intended to represent, to assembly they are just bit patterns, which can always be added, even if doing so makes no sense whatsoever!

Types and Operations

- It's useful to think about which operations are legal for values of each type
 - It doesn't make sense to add a function pointer and an integer in C
 - Makes no sense to add a bit pattern representing a function pointer to one that represents an integer
 - It does make sense to add two integers
 - Makes sense to add two bit patterns that represent integers to get a bit pattern representing sum
 - The problem: both situations have the same assembly language implementation!
 - So can't tell at assembly level which one you're doing⁴⁵

Bottom Line

- If we want to be sure that we only perform operations that “make sense”, then we need
 1. Some sort of type description
 2. Some sort of type system to enforce these distinctions

And to Perhaps Belabor the Point: 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 **only** with the correct types
 - Enforces intended interpretation of values, **because nothing else will!**
 - Once we get down to the machine code level, it's all just 0s and 1s, and the machine is happy to perform whatever operation we want on these, whether or not the operations make sense

Type Checking Overview

- Today, programming languages fall into three categories, with respect to how they treat types:
 - *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, Lisp, Python, Perl)
 - *Untyped*: No type checking (machine code, some would say C)

The Type Wars (Ongoing for Decades)

- Competing views on relative merits of static vs. dynamic typing
- Static typing proponents say:
 - Static checking catches many programming errors at compile time
 - Allows you to prove that some errors can never happen at runtime
 - Avoids overhead of runtime type checks (so faster)
 - Which would have to happen on every operation during execution

The Type Wars (Ongoing for Decades)

- Competing views on relative merits of static vs. dynamic typing
- Dynamic typing proponents say:
 - Static type systems are restrictive
 - Because you have to prove that the program is well-typed, which is accomplished by restricting the kinds of programs you can write
 - Some programs are more difficult to write in a static type system because the compiler has a hard time proving them correct
 - Rapid prototyping difficult within a static type system
 - The belief being that if you're exploring some idea, you may not know what all the types are at that point in time, and having to commit to something that is going to work in all cases (having a type correct program) is constraining and makes the work go slower

The Type Wars (Cont.)

- In practice, code written in statically typed languages usually has an “escape” mechanism
 - Unsafe casts in C, C++, Java
 - In C, results in runtime crash, in Java results in uncaught exception
- People who code in dynamically typed languages often end up retrofitting their code to a statically typed language
 - For optimization, debugging: If a dynamically typed language becomes popular enough, the first thing people want is an optimizing compiler, and the first thing such a compiler needs is type information since it helps generate much better code. So people try to harvest as many “types” here as they can.

The Type Wars (Cont.)

- In practice, code written in statically typed languages usually has an “escape” mechanism
 - Unsafe casts in C, C++, Java
 - In C, results in runtime crash, in Java results in uncaught exception
- People who code in dynamically typed languages often end up retrofitting their code to a statically typed language
 - For optimization, debugging
- It's debatable whether this compromise represents the best or worst of both worlds

Types Outline

- Type concepts in COOL
- Notation for type rules
 - Logical rules of inference
- COOL type rules
- General properties of type systems

Cool Types (Cool is statically typed)

- The types are:
 - Class Names
 - So defining a class defines a new type
 - `SELF_TYPE`
 - More on this later
- The user declares types for identifiers
- The compiler infers types for expressions
 - Infers a type for *every* expression
 - Goes through AST and, using declared type, calculates type for every expression and subexpression

Terminology: Type Checking vs. Type Inference

- *Type Checking* is the process of verifying fully typed programs
 - Simpler: we have a fully typed program (AST with all types filled in)
- *Type Inference* is the process of filling in missing type information
 - Have an AST with no type info or missing type info and we have to fill in missing types
- The two are different (in many languages very 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
I.e., implication statement that some hypothesis implies some conclusion
- 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**
- We'll start with a simplified system and gradually add features
- Building blocks
 - Symbol \wedge is “and”
 - Symbol \Rightarrow is “if-then”
 - $x:T$ is “ x has type T ”

From English to an Inference Rule (2)

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

$(e_1 \text{ has type } \text{Int}) \wedge (e_2 \text{ has type } \text{Int}) \Rightarrow$
 $e_1 + e_2 \text{ has type } \text{Int}$

$(e_1: \text{Int} \wedge e_2: \text{Int}) \Rightarrow (e_1 + e_2): \text{Int}$

Notice that we've gone from English to a purely
mathematical notation

From English to an Inference Rule (3)

The statement

$$(e_1: \text{Int} \wedge e_2: \text{Int}) \Rightarrow e_1 + e_2: \text{Int}$$

is a special case of an inference rule:

$$\text{Hypothesis}_1 \wedge \dots \wedge \text{Hypothesis}_n \Rightarrow \text{Conclusion}$$

Notation for Inference Rules

- By tradition, inference rules are written:

$$\frac{\vdash \text{Hypothesis} \dots \vdash \text{Hypothesis}}{\vdash \text{Conclusion}}$$

Means exactly same thing as the previous slide

- Cool type rules have hypotheses and conclusions

$$\vdash e:T$$

“it is provable that e
has type T ”

- \vdash means “it is provable that ...”

“turnstile” is new notation

Notation for Inference Rules

- By tradition, inference rules are written:

$$\frac{\vdash \text{Hypothesis } \dots \vdash \text{Hypothesis}}{\vdash \text{Conclusion}}$$

- So read: "If it is provable that Hypothesis_1 is true, and it is provable that Hypothesis_2 is true, ..., then it is provable that Conclusion is true"

Some Simple Type Rules

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

$$\frac{\vdash e_1 : \text{Int} \quad \vdash e_2 : \text{Int}}{\vdash e_1 + e_2 : \text{Int}} \quad [\text{Add}]$$

Two Rules (Cont.)

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

Example: Show $1 + 2$ has type Int

$$\frac{\frac{1 \text{ is an int literal}}{\vdash 1 : \text{Int}} \quad \frac{2 \text{ is an int literal}}{\vdash 2 : \text{Int}}}{\vdash 1 + 2 : \text{Int}}$$

Example: Show $1 + 2$ has type Int

proof of the type of number 1

proof of the type of number 2

1 is an int literal

 $\vdash 1 : \text{Int}$

2 is an int literal


 $\vdash 2 : \text{Int}$

$\vdash 1 + 2 : \text{Int}$

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 literal

 $\vdash i: \text{Object}$

Though true, not all that useful, since it doesn't let us perform integer operations on i

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 subexpressions
 - Conclusion is the type of e
- Types are computed in a bottom-up pass over the AST

Rules for Constants

$$\frac{}{\vdash \text{false} : \text{Bool}} \quad [\text{False}]$$
$$\frac{s \text{ is a string literal}}{\vdash s : \text{String}} \quad [\text{String}]$$

Rule for New

`new T` produces an object of type `T`

- Ignore `SELF_TYPE` for now ... (again, we'll deal with this one later)

$$\frac{}{\vdash \text{new } T : T} \quad [\text{New}]$$

Two More Rules

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

$$\frac{\begin{array}{l} \vdash e_1: \text{Bool} \\ \vdash e_2: T \end{array}}{\vdash \text{while } e_1 \text{ loop } e_2 \text{ pool: Object}} \quad [\text{Loop}]$$

e_2 has to have some type here, but we don't care what it is

The Type of a While Loop

That a while loop has type **Object** is something of a design decision

Could have designed a language where the type of the loop is **T** and the value of the loop is the last value of the loop that was executed. Problem: if e_1 is false, you never evaluate e_2 , so type of loop is **void**.

Dereferencing a void gives a runtime error. So to discourage programmers from relying on the loop returning a meaningful value, it is just typed as **Object**.

So far, straightforward. Now, a Problem

- What is the type of a variable reference?

$$\frac{x \text{ is a variable}}{\vdash x: ?} \quad [\text{Var}]$$

- Looking at x by itself, this local, structural rule does not carry enough information to give x a type.

So far, straightforward. Now, a Problem

- Looking at x by itself, this local, structural rule does not carry enough information to give x a type.
 - Inference rules have property that all information needs to be local
 - Everything we need to know to carry out the function of the rule needs to be contained in the rule itself
 - There are no external data structures
 - Nothing being passing around here on the side
 - Everything must be encoded in the rule
 - So far, we just don't know enough to say what the type of a variable should be
 - Solution: Put more information in the rules...

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
 - Ex: in expression x by itself, x is free
 - Ex: in $x + y$, x and y are free
 - Ex: in $\text{let } y \leftarrow \dots \text{ in } x + y$, x is free, but y is not
 - We say y is a *bound* variable in this expression

The Intuition

- If you have an expression with free variables, and you want to type check it, you have know what the types of those variables are.
 - You can type check x if you know the type of x
 - You can type check $x + y$ if you know the types of x and y
 - You can type check the `let` statement if you know the type of x (type of y is provided in statement)
- So the free variables are those variables where you need to know the type in order to carry out the type checking
 - A type environment encodes this information

Type Environments

Let O be a function from **ObjectIdentifiers** to **Types**

We are going to extend the kinds of logical statements that we can prove:

The sentence

$$O \vdash e : T$$

is read: ``Under the assumption that **free** variables have the types given by O , it is provable that the expression e has the type T ''

Type Environments

Let O be a function from **ObjectIdentifiers** to **Types**

We are going to extend the kinds of logical statements that we can prove:

The sentence

$O \vdash e : T$

↑
assumptions

← what we wish to prove

Notation very nicely separates what we are **assuming** from what we are **proving**

Modified Rules

The type environment must be added to all the earlier rules:

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

note this doesn't change anything here because i is an integer literal

$$\frac{\mathcal{O} \vdash e_1 : \text{Int} \quad \mathcal{O} \vdash e_2 : \text{Int}}{\mathcal{O} \vdash e_1 + e_2 : \text{Int}} \quad [\text{Add}]$$

note: same set of assumptions \mathcal{O}

but this does effect change, since it potentially gives type information about free variables in the expressions

New Rules

And we can write new rules (this one fixing our earlier problem)

$$\frac{O(x) = T}{O \vdash x: T} \quad [\text{Var}]$$

Now let's look at a rule that does something interesting with the variables from the point of view of the environments...

Let

$$\frac{O[T_0/x] \vdash e_1 : T_1}{O \vdash \text{let } x:T_0 \text{ in } e_1 : T_1} \text{ [Let-No-Init]}$$

$O[T/y]$ means O modified at the single point y , to return T

(O is a function. $O[T/y]$ is also a function. It is a function that has the same value as O for every input except y , at which it has value T)

I.e., $O[T/y](y) = T$

$O[T/y](x) = O(x)$ for $x \neq y$

Let

$$\frac{O[T_0/x] \vdash e_1 : T_1}{O \vdash \text{let } x:T_0 \text{ in } e_1 : T_1} \quad [\text{Let-No-Init}]$$

So, this says we're going to type check e_1 in the same environment O except that at point x , it's going to have type T_0 , since that is the type of the new identifier that's bound at e_1 .

Note that the **let**-rule enforces variable scope

Let

$$\frac{O[T_0/x] \vdash e_1 : T_1}{O \vdash \text{let } x:T_0 \text{ in } e_1 : T_1} \quad [\text{Let-No-Init}]$$

So, the statement below: We modify our type environment to include a new assumption about x (since the type of x has been determined in the let statement). Once we are out of the let statement, however, this assumption is removed, as we are back in the original type environment.

Note that the **let**-rule enforces variable scope

Notes

- The type environment gives types to the free identifiers in the current scope
 - And is implemented by the symbol table
- The type environment is passed down the AST from the root towards the leaves
- Types are computed up the AST from the leaves towards the root

Let with Initialization

Now consider `let` with initialization:

$$\frac{\begin{array}{c} O \vdash e_0 : T_0 \\ O[T_0/x] \vdash e_1 : T_1 \end{array}}{O \vdash \text{let } x:T_0 \leftarrow e_0 \text{ in } e_1 : T_1} \quad [\text{Let-Init}]$$

Note that we don't use the modified environment for type checking e_0 , since the new definition of x is not available in e_0 , so if there is a declaration of x in e_0 , it must be using an older definition of x .

Let with Initialization

Now consider **let** with initialization:

note this says that
 e_0 has to have same
type as x

$$\frac{\begin{array}{c} O \vdash e_0 : T_0 \\ O[T_0/x] \vdash e_1 : T_1 \end{array}}{O \vdash \text{let } x:T_0 \leftarrow e_0 \text{ in } e_1 : T_1} \quad [\text{Let-Init}]$$

This rule is weak. Why? Because there really is no problem if e_0 has a type that is a subtype of T_0 . But the rule limits only to initializer with same type as x . So we can do better if we introduce subtyping relations on classes.

Subtyping

- Define a relation \leq on classes (subclass relation)
 - $X \leq X$
 - $X \leq Y$ if X inherits from Y
 - $X \leq Z$ if $X \leq Y$ and $Y \leq Z$
- An improvement

$$\frac{\begin{array}{c} O \vdash e_0 : T_0 \\ O[T/x] \vdash e_1 : T_1 \\ T_0 \leq T \end{array}}{O \vdash \text{let } x:T \leftarrow e_0 \text{ in } e_1 : T_1} \quad [\text{Let-Init}]$$

Assignment

- Both **let** rules are sound, but more programs typecheck with the second one
- More uses of subtyping:

$$\frac{\begin{array}{l} O(x) = T_0 \\ O \vdash e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O \vdash x \leftarrow e_1 : T_1} \quad [\text{Assign}]$$

Assignment

- Both **let** rules are sound, but more programs typecheck with the second one
- More uses of subtyping:

$$\frac{\begin{array}{l} O(x) = T_0 \\ O \vdash e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O \vdash x \leftarrow e_1 : T_1} \quad [\text{Assign}]$$

In all of these, look at the hypotheses and the conclusions being drawn. See why the conclusions are reasonable.

Initialized Attributes

- Let $O_C(x) = T$ for all attributes $x:T$ in class C
 - Basically, O_C is the type environment inside class C
- Attribute initialization is similar to **let**, except for the scope of names

$$\frac{\begin{array}{l} O_C(x) = T_0 \\ O_C \vdash e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O_C \vdash x:T_0 \leftarrow e_1 : T_1} \text{ [Attr-Init]}$$

If-Then-Else

- Consider:
if e_0 then e_1 else e_2 fi
- The result can be either e_1 or e_2
- The type is either e_1 's type or e_2 's type
- At compile time, the best we can do is the smallest supertype larger than the type of e_1 or e_2

Least Upper Bounds

- $\text{lub}(X, Y)$, the least upper bound of X and Y , is Z if
 - $X \leq Z \wedge Y \leq Z$
 Z is an upper bound
 - $X \leq Z' \wedge Y \leq Z' \Rightarrow Z \leq Z'$
 Z is least among upper bounds
- In COOL (and in most OO languages), the least upper bound of two types is their least common ancestor in the inheritance tree (think back to algorithms course)

If-Then-Else Revisited

$O \vdash e_0: \text{Bool}$

$O \vdash e_1: T_1$ [If-Then-Else]

$O \vdash e_2: T_2$

$O \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi}: \text{lub}(T_1, T_2)$

Case

- The rule for **case** expressions takes a lub over all branches

$$\begin{array}{l} O \vdash e_0 : T_0 \\ O[T_1/x_1] \vdash e_1 : T_1' \end{array}$$

$$\begin{array}{l} \dots \\ O[T_n/x_n] \vdash e_n : T_n' \end{array} \quad \text{[Case]}$$

$$O \vdash \text{case } e_0 \text{ of } x_1 : T_1 \rightarrow e_1; \dots; x_n : T_n \rightarrow e_n; \text{ esac} : \text{lub}(T_1', \dots, T_n')$$

Method Dispatch

- There is a problem with type checking method calls:

$$\begin{array}{c} O \vdash e_0: T_0 \\ O \vdash e_1: T_1 \\ \dots \\ O \vdash e_n: T_n \\ \hline O \vdash e_0.f(e_1, \dots, e_n): ? \end{array} \quad \text{[Dispatch]}$$

Problem is similar to type checking variable references:
we know nothing here about type of f (what does it return?)

Notes on Dispatch

- Added wrinkle in Cool: method and object identifiers live in different name spaces
 - A method `foo` and an object `foo` can coexist in the same scope
 - So two environments: one for objects, one for methods
- In the type rules, this is reflected by a separate mapping M for method *signatures*

$$M(C, f) = (T_1, \dots, T_n, T_{n+1})$$

means in class C there is a method f

$$f(x_1: T_1, \dots, x_n: T_n): T_{n+1}$$

Note argument types vs. result type

The Dispatch Rule Revisited

$$O, M \vdash e_0: T_0$$
$$O, M \vdash e_1: T_1$$
$$\dots$$
$$O, M \vdash e_n: T_n$$
$$M(T_0, f) = (T_1', \dots, T_n', T_{n+1})$$
$$T_i \leq T_i' \text{ for } 1 \leq i \leq n$$

$$O, M \vdash e_0.f(e_1, \dots, e_n): T_{n+1}$$

[Dispatch]

Note the use of T_0 , which is a type where we would expect a class. What gives here?

Static Dispatch

- Static dispatch is a variation on normal dispatch
- The method is found in the class explicitly named by the programmer
 - Recall from Project 2: This is the dispatch method that uses the @ symbol
- The inferred type of the dispatch expression must conform to the specified type

Static Dispatch (Cont.)

$$O, M \vdash e_0 : T_0$$

$$O, M \vdash e_1 : T_1$$

...

$$O, M \vdash e_n : T_n$$

$$T_0 \leq T$$

[StaticDispatch]

$$M(T, f) = (T_1', \dots, T_n', T_{n+1})$$

$$T_i \leq T_i' \text{ for } 1 \leq i \leq n$$

$$O, M \vdash e_0 @ T.f(e_1, \dots, e_n) : T_{n+1}$$

Run the method not from class of e_0 , but from some ancestor class T of class of e_0

The Method Environment

- The method environment must be added to all rules
- In most cases, M is passed down but not actually used
 - Only the dispatch rules use M

$$\frac{O, M \vdash e_1 : \text{Int} \quad O, M \vdash e_2 : \text{Int}}{O, M \vdash e_1 + e_2 : \text{Int}} \quad [\text{Add}]$$

More Environments

- For some cases involving `SELF_TYPE`, we need to know the class in which an expression appears
 - More on this later
- The full type environment for COOL:
 - A mapping `O` giving types to object id's
 - A mapping `M` giving types to methods
 - The current class `C`

Sentences

The form of a *sentence* in the logic is

$$O, M, C \vdash e : T$$

Example:

$$\frac{O, M, C \vdash e_1 : \text{Int} \quad O, M, C \vdash e_2 : \text{Int}}{O, M, C \vdash e_1 + e_2 : \text{Int}} \quad [\text{Add}]$$

Type Systems

- The rules in this lecture are COOL-specific
 - More info on rules for `self` next time
 - Other languages have very different rules
- General themes
 - Type rules are defined on the structure of expressions
 - Types of variables are modeled by an environment
- Warning: Type rules are very compact!

One-Pass Type Checking

- COOL type checking can be implemented in a single traversal over the AST
 - Though we work down the AST to the leaves, then back up to the root
- Type environment is passed down the tree
 - From parent to child
- Types are passed up the tree
 - From child to parent

Implementing Type Systems

$$\frac{O, M, C \vdash e_1: \text{Int} \quad O, M, C \vdash e_2: \text{Int}}{O, M, C \vdash e_1 + e_2: \text{Int}} \quad [\text{Add}]$$

← record containing O, M, C

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

Note TypeCheck() function is recursive

Let Init

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O[T/x], M, C \vdash e_1 : T_1 \end{array} \quad [Let-Init]}{T_0 \leq T} \quad O, M, C \vdash \text{let } x:T \leftarrow e_0 \text{ in } e_1 : T_1$$

```
TypeCheck(Environment, let x:T ← e0 in e1) = {  
  T0 = TypeCheck(Environment, e0);  
  T1 = TypeCheck(Environment.add(x:T), e1);  
  Check subtype(T0, T1);  
  return T1 }
```