Semantic Analysis
Typechecking in ChocoPy

Lectures 9-11

Outline

- The role of semantic analysis in a compiler
  - A laundry list of tasks
- Scope
- 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 more errors

Errors

- Example 1

```
def f(y: int) -> int:
        return x + 3
```

- Example 2

```
y: str = "abc"
print (y%3)
```

Why a Separate Semantic Analysis?

- Parsing cannot catch some errors

- Some language constructs are not context-free
  - 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 . . . **ChocoPyc** checks:
  1. All identifiers are declared
  2. Types
  3. Inheritance relationships
  4. Classes defined only once
  5. Attributes and 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 ChocoPy !

Scope (Cont.)

- The <u>scope</u> 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 Scope

- Most languages have static scope
  - Scope depends only on the program text, not run-time behavior
  - ChocoPy has static scope

- A few languages are dynamically scoped
  - Lisp, Perl
  - Lisp has changed to mostly static scoping
  - Scope depends on execution of the program

Static Scoping Example

```
x: bool = False
w: str = ""
u: int = 0
# global y  # error
# nonlocal z # error
# global x # error
def f(x: int) -> int:
    global u
    y: int = 1
# print(w) # error
    def g(x: int) -> int:
        nonlocal y
        global u
#       nonlocal x  #error
        y = 9
        print (u)
        print (x)
    print(x)
    print (y)
f(1)
print (x)
print (y)
```

Scope in ChocoPy

- ChocoPy identifier names are introduced by
  - Class declarations
  - Attribute definitions
  - Method definitions
  - Variable declarations
  - Function definitions
  - Formal parameters

Namespace of attributes and methods is different from the rest

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 . . . (Cont.)

- Example: the scope of parameter bindings is one subtree

  def f(x: int) -> object: block

- x can be used in subtree block

Symbol Tables

- Consider again: def f(x: int) -> object: block
- Idea:
  - Before processing block, add definition of x to current definitions, overriding any other definition of x
  - After processing block, remove definition of x and restore old definition of x
- A *symbol table* is a data structure that tracks the current bindings of identifiers

Scope in ChocoPy (Cont.)

- Not all kinds of identifiers follow the most-closely nested rule
- For example, class definitions in ChocoPy
  - Cannot be nested
  - Are *globally visible* throughout the program
- In other words, a class name can be used before it is defined
  - except when you inherit
  - If B inherits A, then A must defined before B

Example: Use Before Definition

```
class Foo (object):
   x: "Bar" = None

class Bar (object):
   ...
```

More Scope (Cont.)

- 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 be redefined (overridden)
  - If they have the same signature in the subclass (except the type of the first parameter)
- Attributes cannot be redefined in a subclass

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

Scopes - Summary

- Scoping rules match uses of identifiers with their declarations
  - Static scoping is the most common form
- Scoping rules can be implemented using symbol tables
  - In one or more passes over the AST

Semantic checks

- Variable, attribute, function, method, and formal parameter names in a scope cannot conflict with each other
- All class names are distinct, and any such name cannot conflict with any other identifier
- nonlocal x: make sure x is defined in an outer scope other than the global scope
- global x: make sure x is defined in the global scope
- If a class A is inherited by B, then A must be defined before B
- If you assign to a variable or use it as the ID in a for loop, then the variable must either be annotated with global or nonlocal, or must be declared explicitly as a local variable in the current scope
- Type int, str, or bool cannot be superclass of any class

Semantic checks

- If a variable is not declared locally, or is not global or nonlocal, then the variable cannot be assigned
- All paths in a method or function must have at most one return statement
  - If a path does not have a return statement, assume that it returns None
  - In a __init__ method, all paths must return None implicitly or explicitly
- A class cannot override an attribute defined in any of its superclass
- The first parameter of any method in a class C must have the type C
- If a method, say m1, overrides a method, say m2, in a super class, then both methods must have the same signature except for the type of the first parameter
- __init__ method must have exactly one formal parameter

Semantic checks

- No return statement unless you are in the body of a method or function

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

Types and Operations

- Most operations are legal only for values of some types
  - 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 the semantic checking rules

Type Checking Overview

- Three kinds of languages:

  - *Statically typed*: All or almost all checking of types is done as part of compilation (C, Java, ChocoPy)

  - *Dynamically typed*: Almost all checking of types is done as part of program execution (Scheme, Python)

  - *Untyped*: No type checking (machine code)

The 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

The Type Wars (Cont.)

- In practice, most code is written in statically typed languages with an "escape" mechanism
  - Unsafe casts in C, native methods in Java, unsafe modules in Modula-3

Type Checking in ChocoPy

Outline

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

ChocoPy Types

- The types are:
  - Class names
  - object, int, str, and bool are builtin class names
  - List of a type
  - Note: there are no base types (as int in Java)
- The user declares types for all identifiers

- The compiler infers types for expressions
  - Infers a type for *every* sub-expression

Type Inference

- Type Checking is the process of checking that the program obeys the type system

- Often involves inferring types for parts of the program
  - Some people call the process type inference when inference is necessary

Rules of Inference

- We have seen two examples of formal notation 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, then 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 $\wedge$ 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

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

$(e_1$ has type int $\wedge$ e_2 has type int$)$ $\Rightarrow$ $e_1 + e_2$ has type int

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 has type int $\wedge$ e_2 has type int) $\Rightarrow$ $e_1 + e_2$ has type int

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

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

$$(\text{Hypothesis}_1 \wedge \ldots \wedge \text{Hypothesis}_n) \Rightarrow \text{Conclusion}$$

This is an inference rule

Notation for Inference Rules

- By tradition inference rules are written

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

- ChocoPy type rules have hypotheses and conclusions of the form:

$$\vdash e : T$$

- $\vdash$ means "we can prove that e has type T"

Two Rules

$$\frac{}{\vdash i : int} \; [int] \quad \text{(i is an integer constant)}$$

Two Rules

$$\frac{}{\vdash i : int} \text{ [int]} \quad \text{(i is an integer constant)}$$

$$\frac{\begin{array}{c}\vdash e_1 : int \\ \vdash e_2 : int\end{array}}{\vdash e_1 + e_2 : int} \text{ [add]}$$

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: 1+2

Example: 1 + 2

$$\dfrac{\dfrac{}{\vdash 1 : int} \qquad \dfrac{}{\vdash 2 : int}}{\vdash 1 + 2 : int}$$

Soundness

- A type system is <u>sound</u> 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:

$$\frac{}{\vdash i : \text{object}} \quad \text{(i is an integer constant)}$$

Type Checking Proofs

- Type checking proves facts $e : T$
  - One type rule is used for each kind of expression

- 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

Rules for Constants

$$\frac{}{\vdash \text{False} : \text{bool}}$$ [bool-false]

$$\frac{}{\vdash \text{True} : \text{bool}}$$ [bool-true]

$$\frac{}{\vdash s : \text{str}}$$ [str] (s is a string constant)

$$\frac{}{\vdash i : \text{int}}$$ [int] (i is an integer constant)

$$\frac{}{\vdash \text{None} : \text{object}}$$ [none]

Arithmetic operations

$$\frac{\vdash e_1 : \text{int} \qquad \vdash e_2 : \text{int}}{\vdash e_1 + e_2 : \text{int}} \; [\text{add}]$$

Same for other operators *, -, //, %

$$\frac{\vdash e : \text{int}}{\vdash - e : \text{int}} \; [\text{negate}]$$

Comparison operations

$$\frac{\begin{array}{c}\vdash e_1 : \text{int} \\ \vdash e_2 : \text{int}\end{array}}{\vdash e_1 < e_2 : \text{bool}} \quad [\text{less}]$$

Same for >=, <=, <, >, ==, !=

Boolean operations

$$\frac{\vdash e_1 : \text{bool} \quad \vdash e_2 : \text{bool}}{\vdash e_1 \text{ and } e_2 : \text{bool}} \text{ [and]}$$

Same for or, ==, !=

$$\frac{\vdash e : \text{bool}}{\vdash \text{not } e : \text{bool}} \text{ [not]}$$

String operations

$$\frac{\vdash e_1 : str \quad \vdash e_2 : str}{\vdash e_1 + e_2 : str} \text{ [str-concat]}$$

$$\frac{\vdash e_1 : str \quad \vdash e_2 : int}{\vdash e_1[e_2] : str} \text{ [str-select]}$$

String operations

$$\frac{\vdash e_1 : str \qquad \vdash e_2 : str}{\vdash e_1 == e_2 : bool} \text{ [str-compare]}$$

$$\frac{\vdash e : str}{\vdash len(e) : int} \text{ [str-len]}$$

Comparison operations non int, str, or bool

$$\frac{\begin{array}{c}\vdash e_1 : T_1 \\ \vdash e_2 : T_2 \\ T_1, T_2 \text{ are not int, str, or bool}\end{array}}{\vdash e_1 \text{ is } e_2 : \text{bool}} \quad [\text{is}]$$

Rule for New (will revisit later)

If T is a class, $T()$ produces an object of type T

$$\frac{}{\vdash T() : T} \quad [new]$$

Notation for Inference Rules for Statements

- By tradition inference rules are written

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

- ChocoPy statements have no type, but they should type check:

$$\vdash s$$

- means "we can prove that s type checks"

If-Then-Else Rule

$$\frac{\begin{array}{c} \vdash e_0 : bool \\ \vdash b_0 \\ \vdash e_1 : bool \\ \vdash b_1 \\ . \\ . \\ \vdash b_n \end{array}}{\vdash \text{if } e_0 \text{: } b_0 \text{ elif } e_1 \text{: } b_1 \ldots \text{ else: } b_n} \quad \text{[if-elif-else]}$$

While Rule

$$\frac{\begin{array}{c}\vdash e : bool \\ \vdash b\end{array}}{\vdash \text{while } e : b}\ [\text{while}]$$

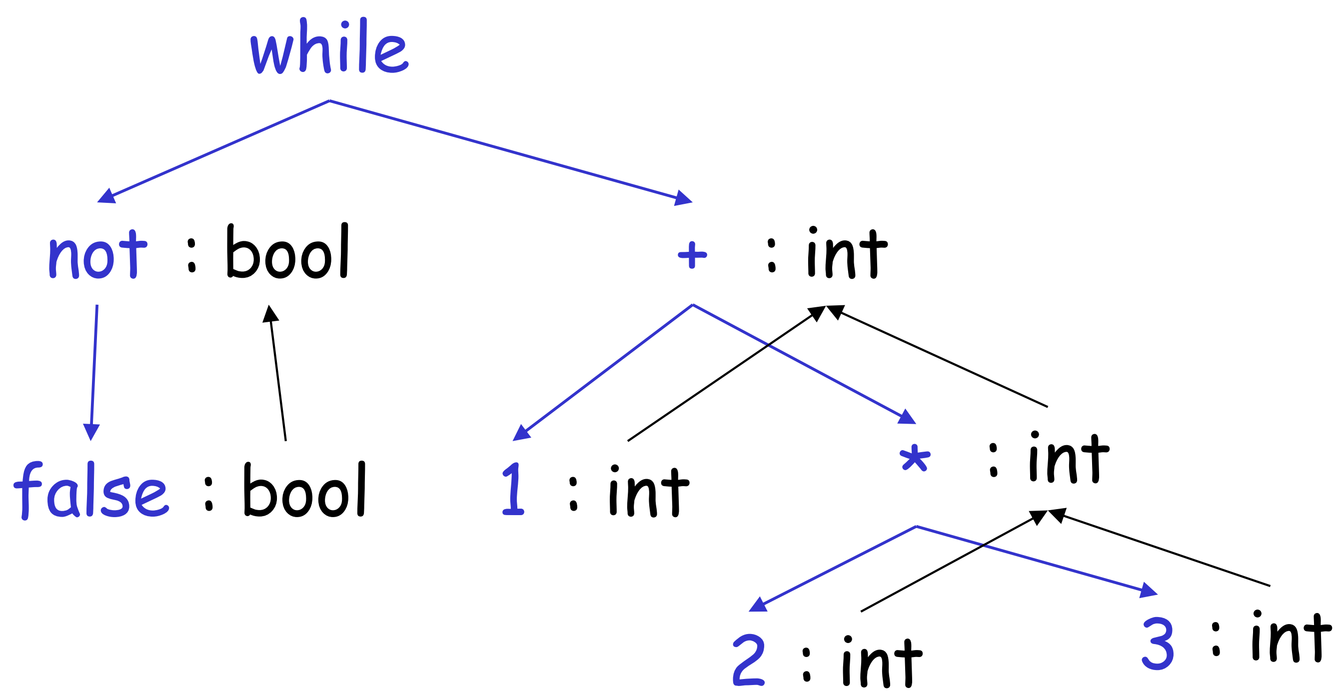
Consider the Rules

$$\frac{\vdash e : \text{bool}}{\vdash \text{not } e : \text{bool}} \quad [\text{not}]$$

$$\frac{\begin{array}{c}\vdash e_1 : \text{bool} \\ \vdash b\end{array}}{\vdash \text{while } e_1 : b} \quad [\text{while}]$$

Typing: Example

- Typing for while not False: 1 + 2 * 3



Typing Derivations

- The typing reasoning can be expressed as an inverted tree:

$$
\dfrac{\dfrac{\vdash \text{False} : \text{bool}}{\vdash \text{not False} : \text{bool}} \qquad \dfrac{\vdash 1 : \text{int} \qquad \dfrac{\vdash 2 : \text{int} \qquad \vdash 3 : \text{int}}{\vdash 2 * 3 : \text{int}}}{\vdash 1 + 2 * 3 : \text{int}}}{\vdash \text{while not False: } 1 + 2 * 3}
$$

- The root of the tree is the statement
- Each node is an instance of a typing rule
- Leaves are the rules with no hypotheses

A Problem

- What is the type of a variable reference?

$$\frac{}{\vdash x : ?} \text{[var-read]} \quad \text{(x is an identifier)}$$

A Problem

- What is the type of a variable reference?

$$\frac{}{\vdash x : ?} \text{[var-read]} \quad \text{(x is an identifier)}$$

- This rules does not have enough information to give a type.
  - We need a hypothesis of the form "*we are in the scope of a declaration of x with type T*")

A Solution: Put more information in the rules!

- A *type environment* gives types for *free* variables
  - A type environment is a mapping from Identifiers to Types
  - A variable is free in an expression if:
    - The expression contains an occurrence of the variable that refers to a declaration outside the expression
  - E.g. in the expression "x", the variable "x" is free
  - E.g. in "def f(x : int) -> int: return x + f(y)" only "y" is free, but "x" and "f" are not

Type Environments and Modified Type Judgement (expressions)

Let O be a function from Identifiers to Types

The sentence $O \vdash e : T$

is read: Under the assumption that variables in the current scope have the types given by O, it is provable that the expression e has the type T

Modified Type Judgement (statements)

Let O be a function from Identifiers to Types

The sentence $O \vdash s$

is read: Under the assumption that variables in the current scope have the types given by O, it is provable that s type checks

The Variable Read Rule

$$\frac{O(id) = T}{O \vdash id : T} \quad \text{[var-read]}$$

Modified Rules

The type environment is added to the earlier rules:

$$\frac{}{O \vdash i : int} \quad [int] \quad (i \text{ is an integer})$$

$$\frac{\begin{array}{c} O \vdash e_1 : int \\ O \vdash e_2 : int \end{array}}{O \vdash e_1 + e_2 : int} \quad [add]$$

While Rule

$$\frac{\begin{array}{c} O \vdash e : \text{bool} \\ O \vdash b \end{array}}{O \vdash \text{while } e \text{: } b} \quad \text{[while]}$$

The Variable Assignment Rule

$$\frac{\begin{array}{c} O(id) = T \\ O \vdash e_1 : T \end{array}}{O \vdash id = e_1 : T} \quad \text{[var-assign]}$$

Weak rule for assignment

- Consider the example:

```
class C (P):
        ...
x : P = None
x = C()
  ...
```

- The previous rule does not allow this code
  - We say that the rule is too weak

Subtyping

- Define a relation $X \leq Y$ on classes to say that:
  - An object of type X could be used when one of type Y is acceptable, or equivalently
  - X conforms with Y
  - In ChocoPy this means that X is a subclass of Y
- Define a relation $\leq$ on classes

  $X \leq X$

  $X \leq Y$ if X inherits from Y

  $X \leq Z$ if $X \leq Y$ and $Y \leq Z$

Expressiveness of Static Type Systems

- A static type system enables a compiler to detect many common programming errors
- The cost is that some correct programs are disallowed
  - Some argue for dynamic type checking instead
  - Others argue for more expressive static type checking

- But more expressive type systems are also more complex

Dynamic And Static Types

- The <u>dynamic type</u> of an object is the class C that is used in the "C()" expression that creates the object
  - A run-time notion
  - Even languages that are not statically typed have the notion of dynamic type
- The <u>static type</u> of an expression is a notation that captures all possible dynamic types the expression could take
  - A compile-time notion

Dynamic and Static Types. (Cont.)

- In early type systems the set of static types correspond directly with the dynamic types
- Soundness theorem: for all expressions E

  dynamic_type(E) = static_type(E)

  (in **all** executions, E evaluates to values of the type inferred by the compiler)

- This gets more complicated in advanced type systems

Dynamic and Static Types in ChocoPy

```
class A:
        ...
class B (A):
        ...
x: A = None;
x = A();
    ...
x = B ();
    ...
```

x has static type *A*

Here, x' s value has dynamic type *A*

Here, x' s value has dynamic type B

- *A* variable of static type *A* can hold values of static type B, if $B \leq A$

Dynamic and Static Types

Soundness theorem for the ChocoPy type system:

$$\forall\, E.\quad \text{dynamic\_type}(E) \le \text{static\_type}(E)$$

Dynamic and Static Types

Soundness theorem for the ChocoPy type system:

$$\forall E. \quad \text{dynamic\_type}(E) \leq \text{static\_type}(E)$$

Why is this Ok?

- For E, compiler uses static_type(E) (call it *C*)
- All operations that can be used on an object of type *C* can also be used on an object of type $C' \leq C$
  - Such as fetching the value of an attribute
  - Or invoking a method on the object
- Subclasses can <u>only add</u> attributes or methods
- Methods can be redefined but with same type !

The Variable Assignment Rule

$$\frac{\begin{array}{c} O(id) = T \\ O \vdash e_1 : T_1 \\ T_1 \leq T \end{array}}{O \vdash id = e_1 : T_1} \quad \text{[var-assign]}$$

The Variable Init Rule

$$\frac{\begin{array}{c} O(id) = T \\ O \vdash e_1 : T_1 \\ T_1 \leq T \end{array}}{O \vdash id: T = e_1} \quad \text{[var-init]}$$

For Rule

$$\frac{\begin{array}{c} O \vdash e : [T_1] \\ O(id) = T \\ O \vdash b \\ T_1 \leq T \end{array}}{O \vdash \text{for id in } e\text{: } b} \quad \text{[for-other]}$$

For Rule for str

$$\frac{\begin{array}{c} O \vdash e : str \\ O(id) = T \\ O \vdash b \\ str \leq T \end{array}}{O \vdash \text{for } id \text{ in } e\text{: } b} \quad \text{[for-str]}$$

List operations

$$\frac{\vdash e : [T]}{\vdash \mathrm{len}(e) : \mathrm{int}} \quad \text{[list-len]}$$

$$\frac{\begin{array}{c}\vdash e_1 : [T] \\ \vdash e_2 : \mathrm{int}\end{array}}{\vdash e_1[e_2] : T} \quad \text{[list-select]}$$

List assignment

$$\frac{\begin{array}{c}\vdash e_1 : [T] \\ \vdash e_2 : \text{int} \\ \vdash e_3 : T_1 \\ T_1 \leq T\end{array}}{\vdash e_1[e_2] = e_3 : T_1} \quad \text{[list-assign]}$$

List operations

$$\frac{\begin{array}{c}\vdash e_1 : T1 \\ \vdash e_2 : T2 \\ \ldots \\ \vdash e_n : Tn\end{array}}{\vdash [e_1, e_2, \ldots, e_n] : [???]} \quad \text{[list-literal]}$$

$$\frac{\begin{array}{c}\vdash e_1 : [T_1] \\ \vdash e_2 : [T_2]\end{array}}{\vdash e_1 + e_2 : [???]} \quad \text{[list-concat]}$$

Least Upper Bounds

- 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 ChocoPy, the least upper bound of two types is their least common ancestor in the inheritance tree

List operations

$$\frac{\begin{array}{c}\vdash e_1 : T1 \\ \vdash e_2 : T2 \\ \ldots \\ \vdash e_n : Tn\end{array}}{\vdash [e_1, e_2, \ldots, e_n] : [lub(T_1, T_2, \ldots T_n)]} \quad \text{[list-literal]}$$

$$\frac{\begin{array}{c}\vdash e_1 : [T_1] \\ \vdash e_2 : [T_2]\end{array}}{\vdash e_1 + e_2 : [lub(T_1, T_2)]} \quad \text{[list-concat]}$$

Function Invocation

- The type information about a function is stored in the type environment O

$$O(f) = \{\$ret{:}T_0,\ x_1{:}T_1,\ \ldots,\ x_n{:}T_n,\ v_1{:}T'_1,\ \ldots,\ v_m{:}T'_m\}$$

  - means function f is of the form

    $f(x_1{:}T_1, \ldots, x_n{:}T_n) \text{ -> } T_0{:}\ \ldots$

  - $v_1, \ldots, v_m$ being the local variables and functions bound in the local scope of f

The Function Invocation Rule

$$\frac{\begin{array}{c} O \vdash e_1 : T''_1 \\ O \vdash e_2 : T''_2 \\ \ldots \\ O \vdash e_n : T''_n \\ O(f) = \{\$ret{:}T_0,\ x_1{:}T_1,\ \ldots,\ x_n{:}T_n,\ v_1{:}T'_1,\ \ldots,\ v_m{:}T'_m\} \\ \text{for } 1 \leq i \leq n{:}\quad T''_i \leq T_i \end{array}}{O \vdash f(e_1, e_2, \ldots, e_n) : T_0} \quad [\text{invoke}]$$

Function Definition Rule

$$\frac{O(f) = \{\$ret{:}T_0, x_1{:}T_1, \ldots, x_n{:}T_n, v_1{:}T'_1, \ldots, v_m{:}T'_m\} \qquad O[T_1/x_1] \ldots [T_n/x_n][T'_1/v_1] \ldots [T'_m/v_m] \vdash b}{O \vdash \text{def } f(x_1{:}T_1, \ldots, x_n{:}T_n) \text{ -> } T_0 : b} \quad \text{[fun-def]}$$

$O[T_0/x]$ means "O modified to map x to T_0 and behaving as O on all other arguments":

$$O[T_0/x]\,(x) = T_0$$
$$O[T_0/x]\,(y) = O(y)$$

Function Rules: Examples

- Consider the following ChocoPy class definitions

```
class A (object): def a(self: A) -> int: return 0
class B (A): def b(self: B) -> int: return 1
```

- An instance of B has methods "a" and "b"
- An instance of A has method "a"
  - A type error occurs if we try to invoke method "b" on an instance of A

Wrong Function Invocation Rule (I)

- Now consider another hypothetical rule:

$$\frac{\begin{array}{c} O \vdash e_1 : T''_1 \\ O(f) = \{\$ret{:}T_0,\ x_1{:}T_1\} \\ T_1 \leq T''_1 \end{array}}{O \vdash f(e_1) : T_0} \quad \text{[invoke]}$$

- How is it different from the correct rule?

Wrong Function Invocation Rule (I)

- Now consider another hypothetical rule:

$$\frac{\begin{array}{c} O \vdash e_1 : T''_1 \\ O(f) = \{\$ret{:}T_0,\ x_1{:}T_1\} \\ T_1 \leq T''_1 \end{array}}{O \vdash f(e_1) : T_0} \quad [invoke]$$

- How is it different from the correct rule?
- The following bad program is well typed

```
def f(x : B) -> int: x.b()
f(A())
```

- Why is this program bad?

Wrong Function Definition Rule (II)

- Now consider a hypothetical rule:

$$\frac{O(f) = \{\$ret{:}T_0,\ x_1{:}T_1\} \qquad O \vdash b}{O \vdash \text{def } f(x_1{:}T_1) \text{ -> } T_0 : b} \quad \text{[fun-def]}$$

- How is it different from the correct rule?

Wrong Function Definition Rule (II)

- Now consider a hypothetical rule:

$$\frac{O(f) = \{\$ret{:}T_0,\ x_1{:}T_1\} \qquad O \vdash b}{O \vdash \text{def } f(x_1{:}T_1) \text{ -> } T_0 : b}\ [\text{fun-def}]$$

- How is it different from the correct rule?
- The following good program does not typecheck

def f(x : int) -> int: return x + 1

Comments

- The typing rules use very concise notation
- They are very carefully constructed
- Virtually any change in a rule either:
  - Makes the type system unsound

    (bad programs are accepted as well typed)
  - Or, makes the type system less usable

    (good programs are rejected)

- But some good programs will be rejected anyway
  - The notion of a good program is undecidable

Return Statement Rule

$$\frac{\begin{array}{c} O \vdash e : T \\ ??? \end{array}}{O \vdash \text{return } e} \quad [\text{return}]$$

Extending Typing Judgement: expressions

The sentence $O, R \vdash e : T$

is read: Under the assumption that variables in the current scope have the types given by O and the return type of current method or function is R, it is provable that the expression e has the type T

Extending Typing Judgement: statements

The sentence $O, R \vdash s$

is read: Under the assumption that variables in the current scope have the types given by O and the return type of current method or function is R, it is provable that the statement s type checks

Function Definition Rule

$$\frac{O(f) = \{\$ret:T_0, x_1:T_1, \ldots, x_n:T_n, v_1:T'_1, \ldots, v_m:T'_m\} \quad O[T_1/x_1, \ldots, T_n/x_n, T'_1/v_1, \ldots, T'_m/v_m], T_0 \vdash b}{O, R \vdash \text{def } f(x_1:T_1, \ldots, x_n:T_n) \text{ -> } T_0 : b}$$

[fun-def]

Return Statement Rule

$$\frac{\begin{array}{c} O, R \vdash e : T \\ T \leq R \end{array}}{O, R \vdash \text{return } e} \quad [\text{return-e}]$$

The Variable Assignment Rule for None

$$\frac{\begin{array}{c} O(id) = T \\ T \text{ is not int, str, or bool} \end{array}}{O, R \vdash id = \text{None} : T} \quad \text{[var-assign-none]}$$

The Variable Init Rule for None

$$\frac{\begin{array}{c} O(id) = T \\ T \text{ is not int, str, or bool} \end{array}}{O, R \vdash id\text{: } T = \text{None}} \quad \text{[var-init-none]}$$

List assignment with None

$$\frac{\begin{array}{c} O, R \vdash e_1 : [T] \\ O, R \vdash e_2 : int \\ T \text{ is not int, str, or bool} \end{array}}{O, R \vdash e_1[e_2] = None: T} \quad \text{[list-assign-none]}$$

The Function Invocation Rule with None

$$
\frac{\begin{array}{c} O, R \vdash e_1 : T''_1 \\ O, R \vdash e_2 : T''_2 \\ \ldots \\ O, R \vdash e_n : T''_n \\ O(f) = \{\$ret{:}T_0, x_1{:}T_1, \ldots, x_n{:}T_n, v_1{:}T'_1, \ldots, v_m{:}T'_m\} \\ \text{for } 1 \leq i \leq n: \\ T''_i \leq T_i \text{ or } (e_i = \text{None and } T_i \text{ is not int, str, or bool}) \end{array}}{O \vdash f(e_1, e_2, \ldots, e_n) : T_0} \quad [\text{invoke}]
$$

Return Statement Rule

$$\frac{\begin{array}{c} O, R \vdash e : T \\ T \leq R \end{array}}{O, R \vdash \text{return } e} \quad \text{[return-e]}$$

$$\frac{R \text{ is not int, str, or bool}}{O, R \vdash \text{return None}} \quad \text{[return-none]}$$

$$\frac{R \text{ is not int, str, or bool}}{O, R \vdash \text{return}} \quad \text{[return]}$$

The Variable Assignment Rule for []

$$\frac{O(id) = [T]}{O, R \vdash id = [] : [T]} \quad \text{[var-assign-nil]}$$

List assignment with []

$$\frac{\begin{array}{c} O, R \vdash e_1 : [[T]] \\ O, R \vdash e_2 : int \end{array}}{O, R \vdash e_1[e_2] = []: [T]} \text{ [list-assign-nil]}$$

The Function Invocation Rule with None and []

$$\dfrac{\begin{array}{c} O, R \vdash e_1 : T''_1 \\ O, R \vdash e_2 : T''_2 \\ \ldots \\ O, R \vdash e_n : T''_n \\ O(f) = \{\$ret{:}T_0, x_1{:}T_1, \ldots, x_n{:}T_n, v_1{:}T'_1, \ldots, v_m{:}T'_m\} \\ \text{for } 1 \le i \le n: \\ T''_i \le T_i \text{ or } (e_i = \text{None and } T_i \text{ is not int, str, or bool}) \text{ or} \\ (e_i = [\,] \text{ and } T_i \text{ is a list type}) \end{array}}{O, R \vdash f(e_1, e_2, \ldots, e_n) : T_0} \quad [\text{invoke}]$$

Return Statement Rule

$$\frac{O, R \vdash e : T \qquad T \leq R}{O, R \vdash \text{return } e} \text{ [return-e]}$$

$$\frac{}{O, [T] \vdash \text{return } []} \text{ [return-none]}$$

Method Dispatch

- In ChocoPy, methods and attributes live in different name spaces than variable identifiers, class names, and function names
- In the type rules, this is reflected by a separate mapping M for method signatures and attribute types

$$M(C,f) = \{\$ret:T_0,\ x_1:T_1,\ \ldots,\ x_n:T_n,\ v_1:T'_1,\ \ldots,\ v_m:T'_m\}$$

  - means in class C there is a method f
  - $f(x_1:T_1, \ldots, x_n:T_n)$ -> T_0: ...
  - $v_1, \ldots, v_m$ being the local variables and functions defined in the top-level scope of f

$$M(C,a) = T$$

  - means in class C there is an attribute a of type T

An Extended Typing Judgment

- Now we have two environments O and M
- The form of the typing judgment for expressions is

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

read as: "with the assumption that the variable identifiers have types as given by O and the method/attribute identifiers have signatures as given by M, the expression e occurring in the body of class C and method/function whose return type is R has type T"

- The form of the typing judgment for statements is

$$O, M, C, R \vdash s$$

read as: "with the assumption that the variable identifiers have types as given by O and the method/attribute identifiers have signatures as given by M, the statement s occurring in the body of C and method/function whose return type is R type checks"

The Method/Attribute Environment

- The method/attribute environment must be added to all rules
- In most cases, M is passed down but not actually used
  - Example of a rule that does not use M:

$$\frac{O, M, C, R \vdash e_1 : \text{int} \qquad O, M, C, R \vdash e_2 : \text{int}}{O, M, C, R \vdash e_1 + e_2 : \text{int}} \quad [\text{add}]$$

  - Only the dispatch and attribute related rules use M

The Attribute Read Rule

$$\frac{\begin{array}{c} O, M, C, R \vdash e_1 : T_1 \\ M(T_1, id) = T_0 \end{array}}{O, M, C, R \vdash e_1.id : T_0} \quad \text{[attr-read]}$$

The Attribute Init Rule

$$\frac{\begin{array}{c} M(C, id) = T \\ O, M, C, R \vdash e_1 : T_1 \\ T_1 \leq T \end{array}}{O, M, C, R \vdash id{:}\ T = e_1} \quad \text{[attr-init]}$$

Similarly add rules for None

The Attribute Assignment Rule

$$\frac{\begin{array}{c} O, M, C, R \vdash e_0 : T_0 \\ M(T_0, id) = T \\ O, M, C, R \vdash e_1 : T_1 \\ T_1 \leq T \end{array}}{O, M, C, R \vdash e_0.id = e_1 : T_1} \quad \text{[attr-assign]}$$

Similarly add rules for None

The Method Dispatch Rule

$$\frac{\begin{array}{c} O, M, C, R \vdash e_1 : T''_1 \\ O, M, C, R \vdash e_2 : T''_2 \\ \ldots \\ O, M, C, R \vdash e_n : T''_n \\ M(T''_1, f) = \{\$ret{:}T_0, x_1{:}T_1, \ldots, x_n{:}T_n, v_1{:}T'_1, \ldots, v_m{:}T'_m\} \\ T''_1 \leq T_1 \\ \text{for } 2 \leq i \leq n: \\ T''_i \leq T_i \ \text{ or } (e_i = \text{None and } T_i \text{ is not int, str, or bool}) \end{array}}{O, M, C, R \vdash e_1.f(e_2,\ldots,e_n) : T_0} \quad [\text{dispatch}]$$

Method Definition Rule

$$\frac{\begin{array}{c} M(C, f) = \{\$ret{:}T_0, x_1{:}T_1, \ldots, x_n{:}T_n, v_1{:}T'_1, \ldots, v_m{:}T'_m\} \\ O[T_1/x_1, \ldots, T_n/x_n, T'_1/v_1, \ldots, T'_m/v_m], M, C, T_0 \vdash b \\ C = T_1 \end{array}}{O, M, C, R \vdash f(x_1{:}T_1, \ldots, x_n{:}T_n) \text{ -> } T_0 : b}$$

[method-def]

__init__ Definition Rule

$$\frac{\begin{array}{c} M(C, \_\_init\_\_) = \{\$ret{:}T_0, x_1{:}T_1, v_1{:}T'_1, \ldots, v_m{:}T'_m\} \\ O[T_1/x_1, T'_1/v_1, \ldots, T'_m/v_m], M, C, C \vdash b \\ C = T_1 \end{array}}{O, M, C, R \vdash \text{def } \_\_init\_\_(x_1{:}T_1) \text{ -> } C : b}$$

[init-def]

Type Systems

- The rules in these lecture were ChocoPy-specific
  - 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
- Types are a play between flexibility and safety