Type Checking

Lecture 19

(from notes by G. Necula)

Administrivia

- Test run this evening around midnight
- Test is next Wednesday at 6 in 306 Soda
- Please let me know soon if you need an alternative time for the test.
- Please use bug-submit to submit problems/questions
- Review session Sunday in 310 Soda 4-6PM

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

Why Do We Need Type Systems?

Consider the assembly language fragment

```
addi  $r1, $r2, $r3
```

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

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

What Can Types do For Us?

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

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

```
class Client {
   f(fs : FileSystem) {
       File fdesc <- fs.open("foo")
       …
   } -- 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)

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 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

(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

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

- Type rules have hypotheses and conclusions of the form:

$$\vdash e : T$$

- $\vdash$ means “we can prove that . . .”

Two Rules

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

$$\frac{\begin{array}{c}\vdash e_1 : \mathrm{Int} \\ \vdash e_2 : \mathrm{Int}\end{array}}{\vdash e_1 + e_2 : \mathrm{Int}} \quad [\mathrm{Add}]$$

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
- We can fill the template with ANY expression!
- Logic nerds: Why is the following correct?

$$\frac{\vdash \text{true} : \text{Int} \qquad\qquad \vdash \text{false} : \text{Int}}{\vdash \text{true} + \text{false} : \text{Int}}$$

Example: 1 + 2

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

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; here's one that's not very useful:

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

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

$$\frac{}{\vdash \text{False} : \text{Bool}} \quad [\text{Bool}]$$

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

Object Creation Example

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

(T denotes a class with parameterless constructor)

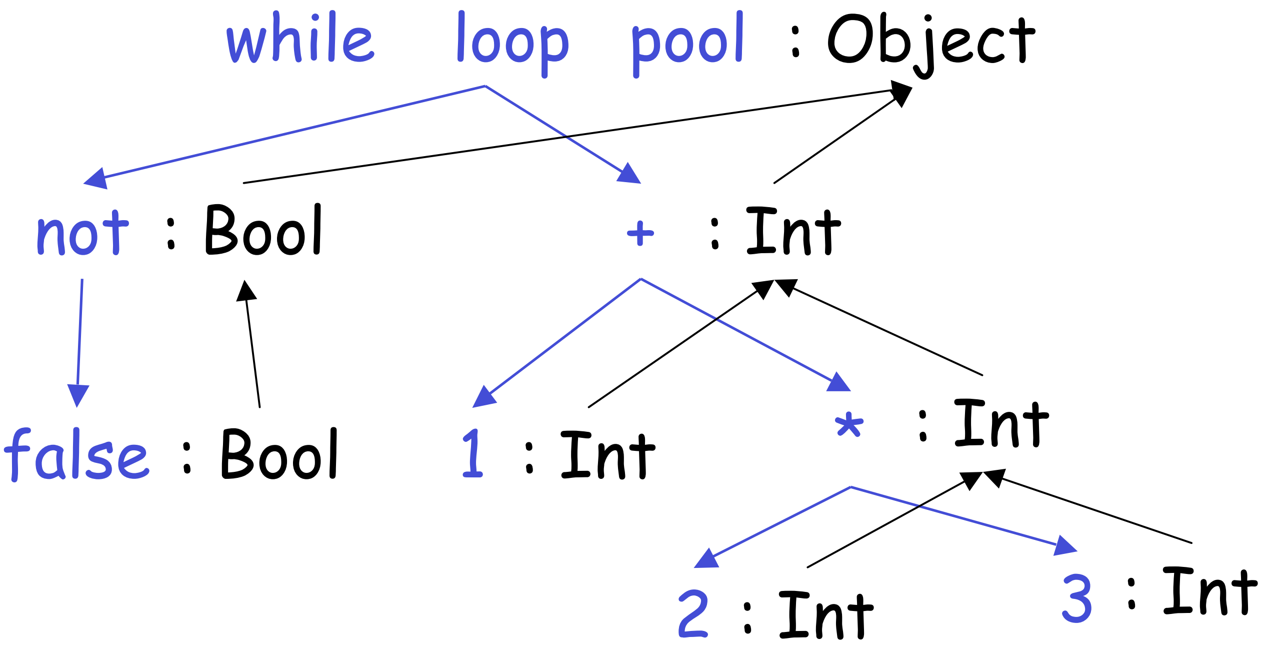
Two More Rules (Not From Pyth)

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

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

Typing: Example

- Typing for while not false loop 1 + 2 * 3 pool



Typing Derivations

- The typing reasoning can be expressed as a tree:

$$\dfrac{\dfrac{\text{|- false : Bool}}{\text{|- not false : Bool}} \qquad \dfrac{\text{|- 1 : Int} \qquad \dfrac{\text{|- 2 : Int} \qquad \text{|- 3 : Int}}{\text{|- 2 * 3 : Int}}}{\text{|- 1 + 2 * 3: Int}}}{\text{|- while not false loop 1 + 2 * 3 : Object}}$$

- The root of the tree is the whole expression
- 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?

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

- 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 "(lambda (x) (+ x y))" only "y" is free
  - E.g. in "(+ x (lambda (x) (+ x y))" both "x" and "y" are free

Type Environments

Let O be a function from Identifiers to Types

The sentence O |- 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 Rules

The type environment is added to the earlier rules:

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

$$\frac{\begin{array}{c} O \vdash e_1 : \text{Int} \\ O \vdash e_2 : \text{Int} \end{array}}{O \vdash e_1 + e_2 : \text{Int}} \quad [\text{Add}]$$

New Rules

And we can write new rules:

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

Lambda (from Python)

$$\frac{O[Any/x] \mid\!- e_1 : T_1}{O \mid\!- \text{lambda } x\text{: } e_1 : Any \to T_1} \quad [Lambda]$$

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

$O[Any/x]\,(x) = Any$

$O[Any/x]\,(y) = O(y)$, x and y distinct

Let (From the COOL Language)

- Let statement creates a variable x with given type T_0 that is then defined throughout e_1

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

Let. Example.

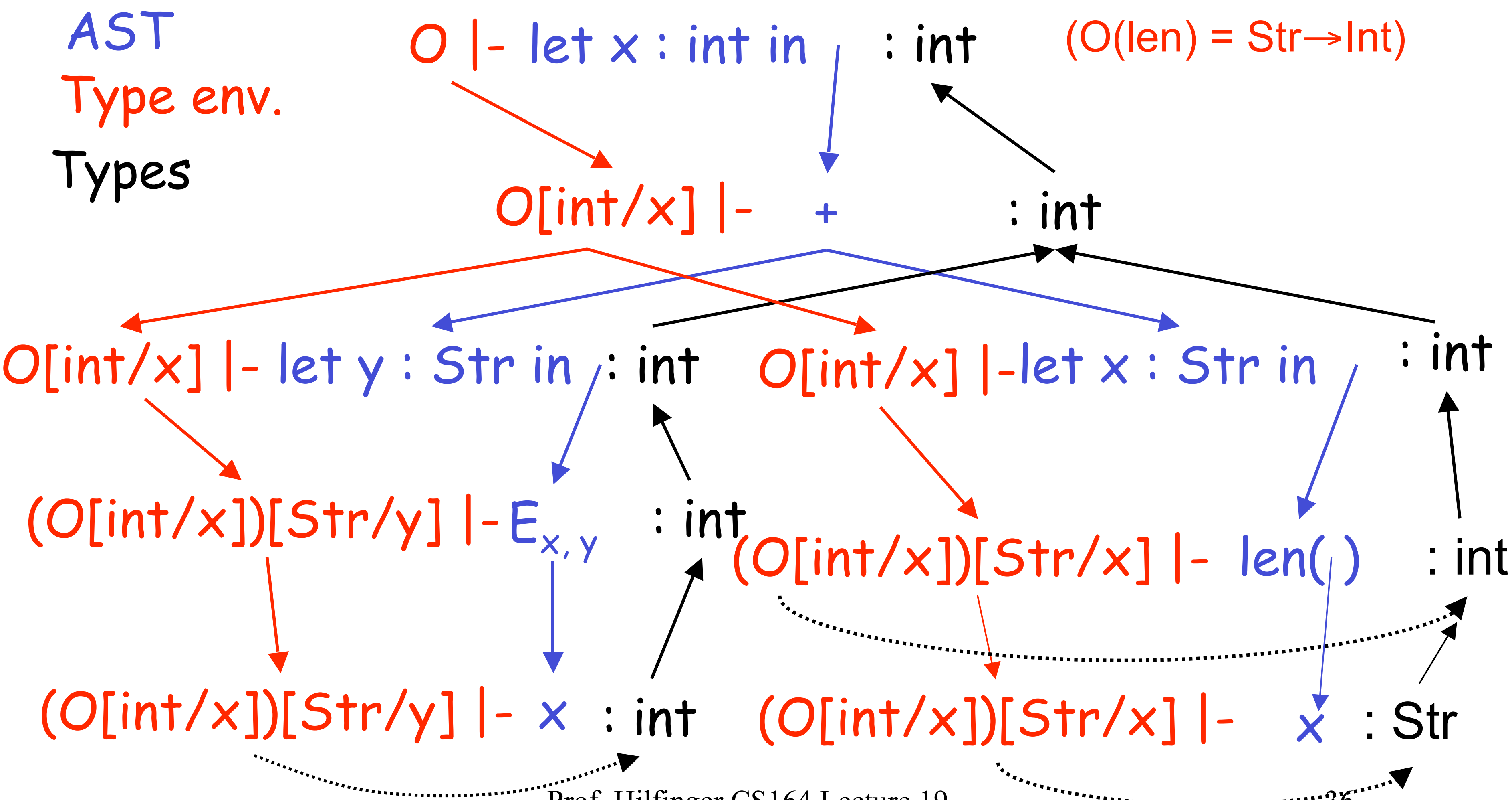
- Consider the Cool expression

  let x : T_0 in (let y : T_1 in $E_{x,\,y}$) + (let x : T_2 in $F_{x,\,y}$)

  (where $E_{x,\,y}$ and $F_{x,\,y}$ are some Cool expression that contain occurrences of "x" and "y")

- Scope
  - of "y" is $E_{x,\,y}$
  - of outer "x" is $E_{x,\,y}$
  - of inner "x" is $F_{x,\,y}$
- This is captured precisely in the typing rule.

Let Example.



AST

Type env.

Types

O |- let x : int in : int

(O(len) = Str→Int)

O[int/x] |- + : int

O[int/x] |- let y : Str in : int

O[int/x] |-let x : Str in : int

(O[int/x])[Str/y] |- $E_{x, y}$: int

(O[int/x])[Str/x] |- len( ) : int

(O[int/x])[Str/y] |- x : int

(O[int/x])[Str/x] |- x : Str

Notes

- The type environment gives types to the free identifiers in the current scope
- 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

COOL also has a let with initialization (I'll let you guess what it's supposed to mean):

$$\frac{O \vdash e_0 : T_0 \qquad O[T_0/x] \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1} \text{[Let-Init]}$$

This rule is weak (i.e. too conservative). Why?

Let with Initialization

- Consider the example:

```
class C inherits P { … }
…
let x : P ← new C in …
 …
```

- The previous let 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 Cool 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 $Y \leq Z$

Let with Initialization (Again)

$$\frac{O \mid\!- \; e_0 : T \qquad T \leq T_0 \qquad O[T_0/x] \mid\!- \; e_1 : T_1}{O \mid\!- \; \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1} \text{[Let-Init]}$$

- Both rules for let are sound
- But more programs type check with the latter

Let with Subtyping. Notes.

- There is a tension between
  - Flexible rules that do not constrain programming
  - Restrictive rules that ensure safety of execution

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

Dynamic and Static Types. (Cont.)

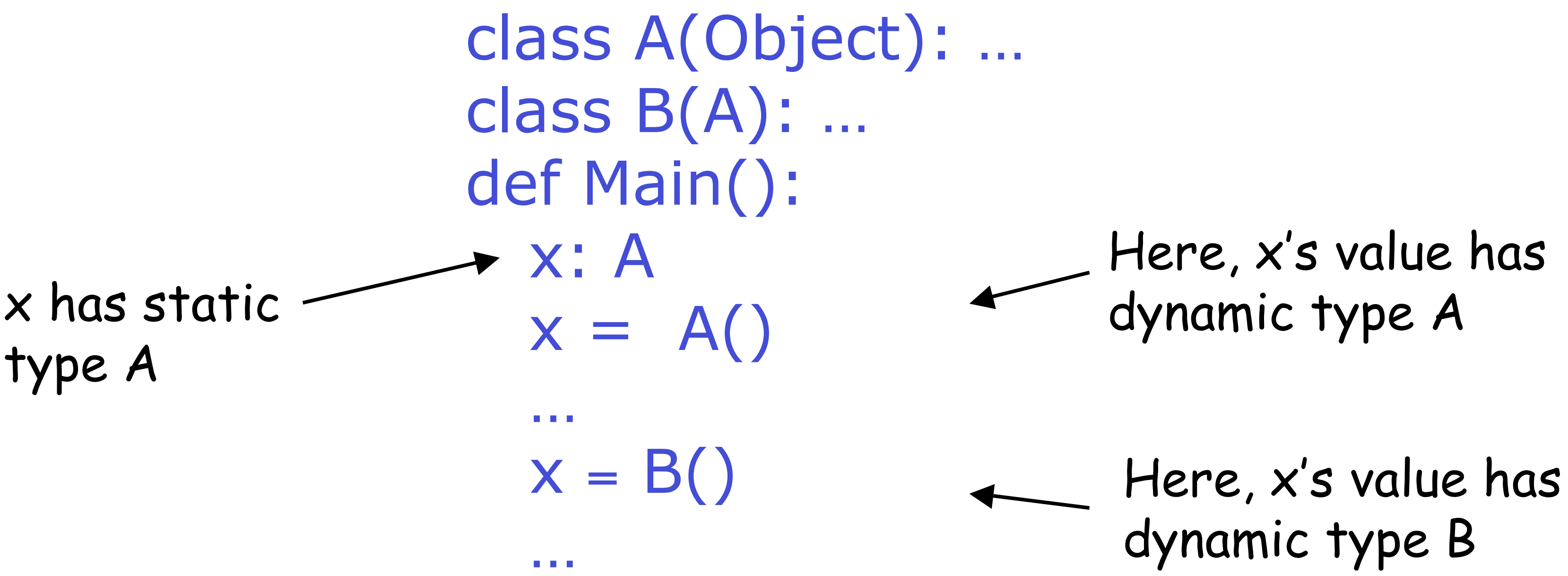
- In early type systems the set of static types correspond directly with the dynamic types
- Soundness theorem: for all expressions E

  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



```
class A(Object): …
class B(A): …
def Main():
  x: A
  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 ≤ A

Dynamic and Static Types

Soundness theorem:

$$\forall E. \quad dynamic\_type(E) \le static\_type(E)$$

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 $C' \le C$
  - Such as fetching the value of an attribute
  - Or invoking a method on the object
- Subclasses can *only add* attributes or methods
- Methods can be redefined but with same type !

Let. Examples.

- Consider the following Cool class definitions

```
Class A { a() : Int { 0 }; }
Class B inherits A { b() : Int { 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

Example of Wrong Let Rule (1)

- Now consider a hypothetical let rule:

$$\frac{O \vdash e_0 : T \qquad T \leq T_0 \qquad O \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1}$$

- How is it different from the correct rule?

- The following good program does not typecheck

```
let x : Int ← 0 in x + 1
```

- And some bad programs do typecheck

```
foo(x : B) : Int { let x : A ← new A in A.b() }
```

Example of Wrong Let Rule (2)

- Now consider another hypothetical let rule:

$$\frac{O \vdash e_0 : T \qquad T_0 \leq T \qquad O[T_0/x] \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1}$$

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

let x : B ← new A in x.b()

- Why is this program bad?

Example of Wrong Let Rule (3)

- Now consider another hypothetical let rule:

$$\frac{O \mid\!\!- e_0 : T \qquad\qquad T \leq T_0 \qquad\qquad O[T/x] \mid\!\!- e_1 : T_1}{O \mid\!\!- \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T_1}$$

- How is it different from the correct rule?

- The following good program is not well typed

  let x : A ← new B in {… x ← new A; x.a(); }

- Why is this program not well typed?

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

Assignment

More uses of subtyping: To the left, rule for languages with assignment *expressions;* to the right, assignment *statements*

$$\frac{O(id) = T_0 \quad O \vdash e_1 : T_1 \quad T_1 \leq T_0}{O \vdash id \leftarrow e_1 : T_1}$$

$$\frac{O(id) = T_0 \quad O \vdash e_1 : T_1 \quad T_1 \leq T_0}{O \vdash id \leftarrow e_1 : void}$$

Assignment in Pyth

- Pyth rule is looser than most.
- Doesn't by itself guarantee runtime type correctness, so check will be needed in some cases.

$$\frac{O(id) = T_0 \qquad O \vdash e_1 : T_1 \qquad T_1 \leq T_0 \vee T_0 \leq T_1}{O \vdash id \leftarrow e_1 : Void}$$

Function call in Pyth

- Parameter passing resembles assignment
- Taking just the single-parameter case:

$$\frac{\begin{array}{c} O \mid\!- \; e_0 : T_1 \rightarrow T_2 \\ O \mid\!- \; e_1 : T_3 \\ T_1 \leq T_3 \vee T_3 \leq T_1 \end{array}}{O \mid\!- \; e_0 (e_1) : T_2}$$

Conditional Expression

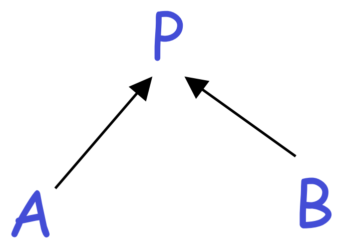
- Consider:

  if e_0 then e_1 else e_2 fi   or e_0 ? e_1 : e_2 in C

- The result can be either e_1 or e_2
- The dynamic type is either e_1's or e_2's type
- The best we can do statically is the smallest supertype larger than the type of e_1 and e_2

If-Then-Else example

- Consider the class hierarchy



- … and the expression

  if … then new A else new B fi

- Its type should allow for the dynamic type to be both A or B
  - Smallest supertype is P

Least Upper Bounds

- $\mathrm{lub}(X,Y)$, 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

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

If-Then-Else Revisited

$$\frac{\begin{array}{c} O \mid\!- \; e_0 : \text{Bool} \\ O \mid\!- \; e_1 : T_1 \\ O \mid\!- \; e_2 : T_2 \end{array}}{O \mid\!- \; \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi} : \text{lub}(T_1, T_2)} \quad \text{[If-Then-Else]}$$