Type Checking

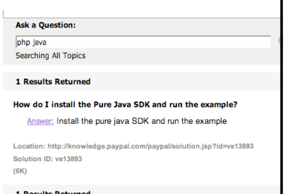
#1

New Lecture Style

- Response to suggestion: today I will pause for five seconds at the end of every slide.
- Think about whether or not you like this.
- If I fail to pause at the end of a slide you should jeer me with great gusto.

#2

Passing Out Review Forms



#3

One-Slide Summary

- A **type environment** gives types for free variables. You typecheck a let-body with an environment that has been **updated** to contain the new let-variable.
- If an object of type X could be used when one of type Y is acceptable then we say X is a **subtype** of Y, also written $X \leq Y$.
- A type system is **sound** if $\forall$ E. dynamic_type(E) $\leq$ static_type(E)

#4

Lecture Outline

- Typing Rules
- Typing Environments
- "Let" Rules
- Subtyping
- Wrong Rules

#5

Example: 1 + 2

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

#6

Soundness

- A type system is **sound** if
  - Whenever ⊢ e : T
  - Then e evaluates to a value of type T
- We only want sound rules
  - But some sound rules are better than others:

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

#7

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 itself

#8

Rules for Constants

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

$$\frac{}{\vdash s : \text{String}} \text{[String]}$$

(s is a string constant)

#9

Rule for New

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

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

#10

Two More Rules

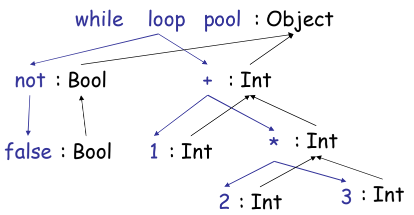
$$\frac{\vdash e : \text{Bool}}{\vdash \text{not } e : \text{Bool}}\ [\text{Not}]$$

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

#11

Typing: Example

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



#12

Typing Derivations

- The typing reasoning can be expressed as a tree:

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

- 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

#13

A Problem

- What is the type of a variable reference?

$$\dfrac{}{\vdash x : ?}\,[\text{Var}] \qquad (x \text{ is an identifier})$$

- The local, structural rule does *not* carry enough information to give x a type.

#14

A Solution: Put more information in the rules!

- A **type environment** gives types for **free** variables
  - A **type environment** is a mapping from Object_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
  - in the expression "x", the variable "x" is free
  - in "let x : Int in x + y" only "y" is free
  - in "x + let x : Int in x + y" both "x", "y" are free

#15

Type Environments

Let O be a function from Object_Identifiers to Types

The sentence $O \vdash e : T$

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

#16

Modified Rules

The type environment is added to the earlier rules:

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

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

#17

New Rules

And we can write new rules:

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

Equivalently:

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

#18

Let

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

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

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

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

#19

Let Example

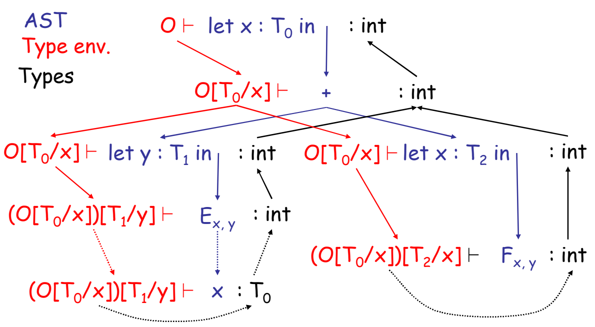
- Consider the Cool expression

  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.

#20

Let. Example.



#21

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

#22

Let with Initialization

Now consider let with initialization:

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

This rule is weak. Why?

#23

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** or **incomplete**

#24

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 $Y \leq Z$

#25

Let With Initialization (Better)

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

- Both rules for let are sound
- But more programs type check with this new rule (it is more complete)

#26

Type System Tug-of-War

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

#27

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

#28

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 notation that captures all possible dynamic types the expression could take
  - A **compile-time** notion

#29

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 (e.g., Java, Cool)

#30

Dynamic and Static Types in COOL

```
class A { … }
class B inherits A {…}
class Main {
    A x ← new A;
    …
    x ← new B;
    …
}
```

x has static type A → `A x ← new A;` ← Here, x's value has dynamic type A

`x ← new B;` ← Here, x's value has dynamic type B

- A variable of static type A can hold values of static type B, if B ≤ A

#31

Dynamic and Static Types

Soundness theorem for the Cool type system:

$\forall$ E. dynamic_type(E) ≤ static_type(E)

Why is this Ok?

- For E, compiler uses static_type(E)
- All operations that can be used on an object of type C can also be used on an object of type C' ≤ 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 the same types!

#32

Subtyping Example

- 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

#33

Example of Wrong Let Rule (1)

- Now consider a hypothetical wrong 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`
- Why?

#34

Example of Wrong Let Rule (2)

- Now consider a hypothetical wrong 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?

#35

Example of Wrong Let Rule (3)

- Now consider a hypothetical wrong let rule:

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

- How is it different from the correct rule?
- The following good program is not well typed
  `let x : A ← new B in {… x ← new A; x.a(); }`
- Why is this program not well typed?

#36

Typing Rule Notation

- 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 (**incomplete**) (good programs are rejected)
- But some good programs will be rejected anyway
  - The notion of a good program is **undecidable**

#37

Assignment

More uses of subtyping:

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

#38

Initialized Attributes

- Let $O_C(x) = T$ for all attributes $x{:}T$ in class C
  - O_C represents the class-wide scope
- Attribute initialization is similar to let, except for the scope of names

$$\frac{O_C(id) = T_0 \quad O_C \vdash e_1 : T_1 \quad T_1 \leq T_0}{O_C \vdash id : T_0 \leftarrow e_1 \;;} \; [Attr\text{-}Init]$$

#39

If-Then-Else

- Consider:
  if e_0 then e_1 else e_2 fi

- The result can be either e_1 or e_2

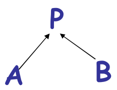
- The dynamic type is either e_1's or e_2's type

- The best we can do statically is the **smallest supertype** larger than the type of e_1 and e_2

#40

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

#41

Least Upper Bounds

- Define: **lub(X,Y)** to be the **least upper bound** of X and Y. lub(X,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, the least upper bound of two types is their **least common ancestor** in the **inheritance tree**

#42

If-Then-Else Revisited

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

[If-Then-Else]

#43

Case

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

$$\frac{O \vdash e_0 : T_0 \quad O[T_1/x_1] \vdash e_1 : T_1' \quad \ldots \quad O[T_n/x_n] \vdash e_n : T_n'}{O \vdash \text{case } e_0 \text{ of } x_1{:}T_1 \Rightarrow e_1; \ldots; x_n : T_n \Rightarrow e_n; \text{ esac} : lub(T_1',\ldots,T_n')}$$

[Case]

#44

Next Time (Post-Midterm)

- Type checking method dispatch
- Type checking with SELF_TYPE in COOL

#45

Homework

- Today: WA3 due
- Friday: PA3 due
  - Parsing!
- **Tuesday Feb 27 - Midterm 1 in Class**
  - **9:35 - 10:40**
  - **One page of notes (front and back) hand-written by you**
- Next Thursday: Read Chapter 7.2