Type Checking

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

Lecture Outline

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

Example: 1 + 2

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

If we can prove it, then it's true!

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 worse than others:

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

THE BASS FROM THAT CAR IS DRIVING ME NUTS.
THUMPA THUMPA
ME TOO. GIVE ME A HAND HERE.
I'M AFRAID TO ASK.
THIS SYSTEM DETECTS BASS RHYTHMS AND FLOODS THE TARGET WITH A PHASE-SHIFTED REPLICA SIGNAL.
THE RESONANCE SHOULD BLOW OUT THEIR SPEAKERS.
THUMPA THUMPA
BLAM
SPEAKERS DOWN. NOW FLIP THAT RED SWITCH.
CLICK
SHIRLEY SHIRLEY BO BIRLEY BANANA FANNA FO FIRLEY
YOU'RE HORRIFYING.
OKAY, NOW THROW THE SWITCH LABELED "MACARENA"

Type Checking Proofs

- Type checking proves facts like 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

Rules for Constants

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

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

(s is a string constant)

Rule for New

new T produces an object of type T

- Ignore SELF_TYPE for now . . .

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

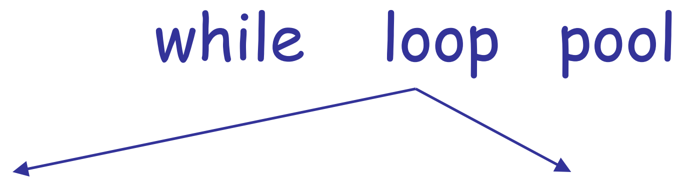
Two More Rules

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

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

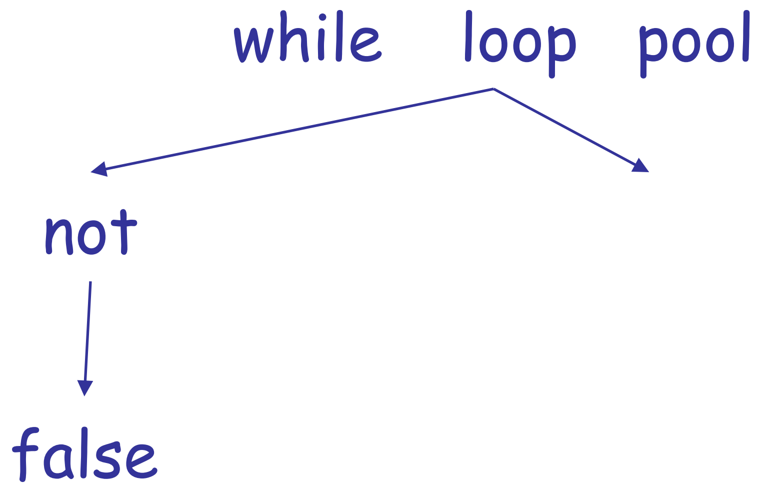
Typing: Example

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



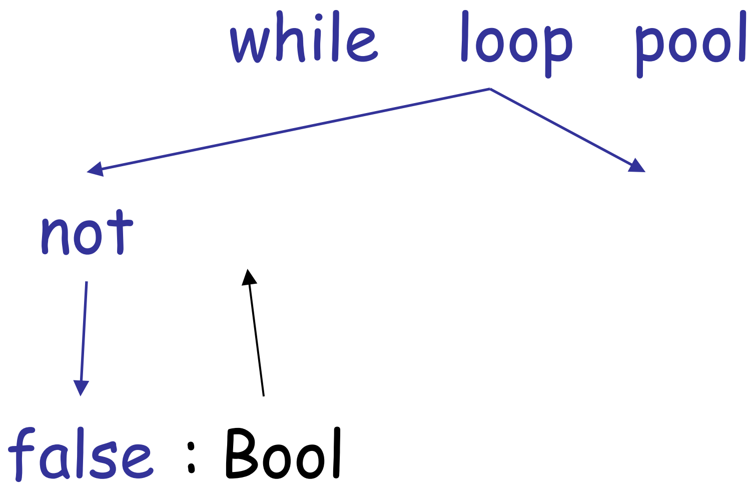
Typing: Example

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



Typing: Example

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



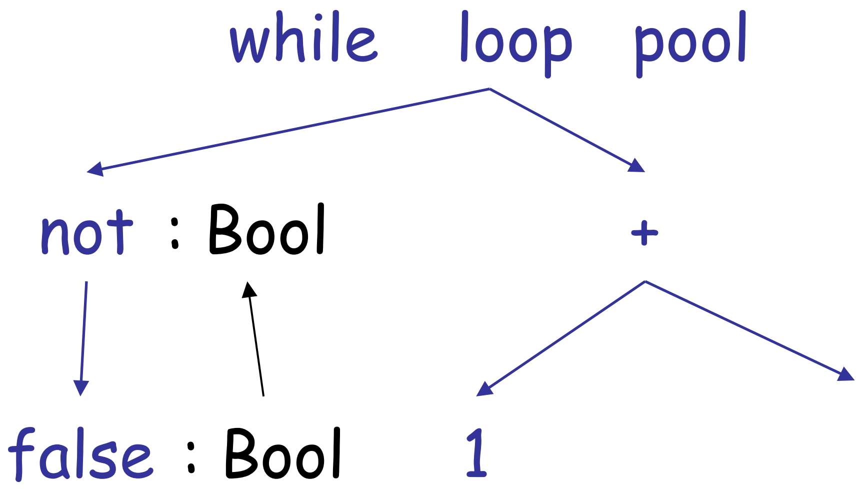
Typing: Example

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

$$\begin{tikzcd}
 & & \text{while} \quad \text{loop} \quad \text{pool} \arrow[dll] \arrow[dr] & \\
\text{not} \arrow[d] & : \text{Bool} & & + \\
\text{false} & : \text{Bool} \arrow[u] & &
\end{tikzcd}$$

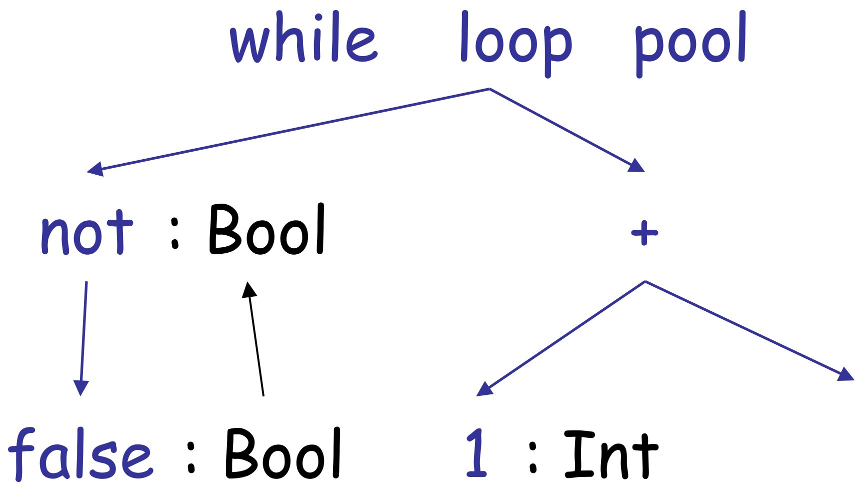
Typing: Example

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



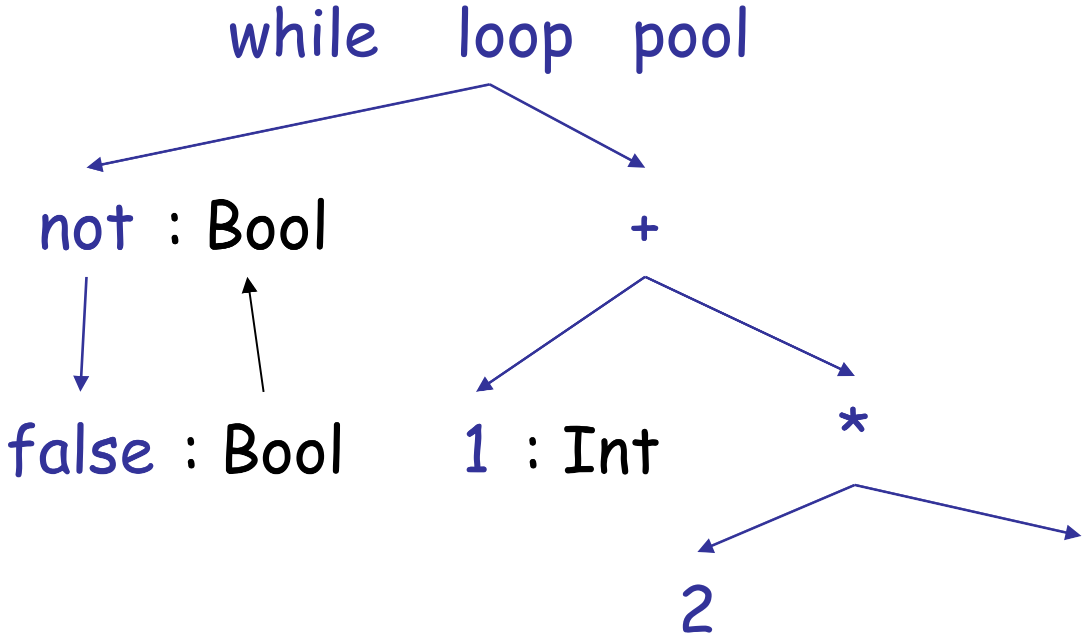
Typing: Example

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



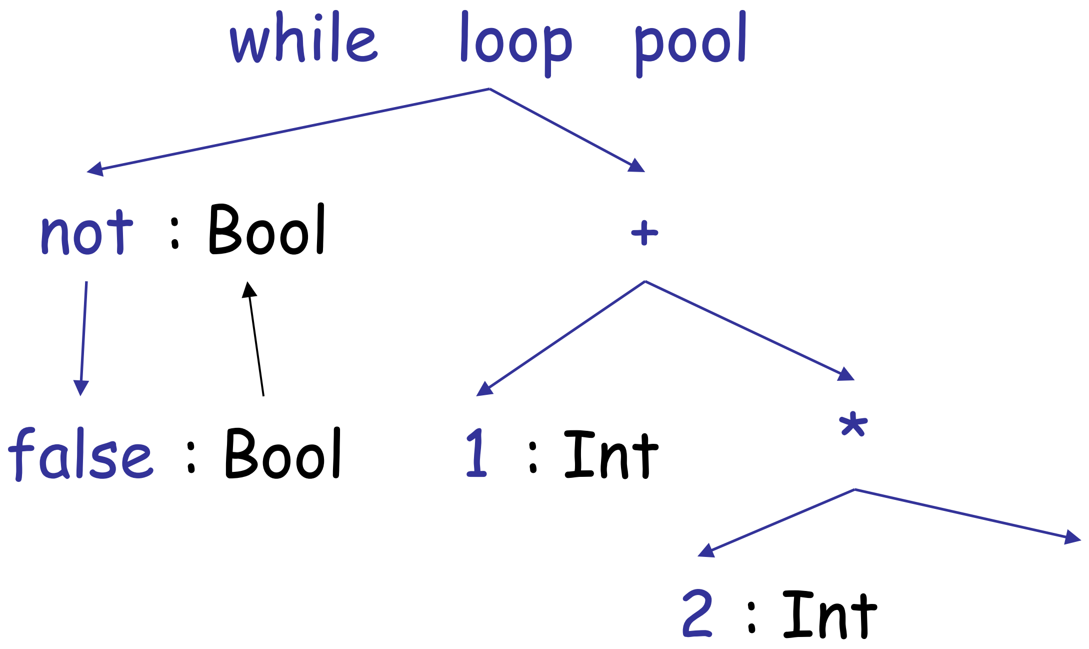
Typing: Example

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



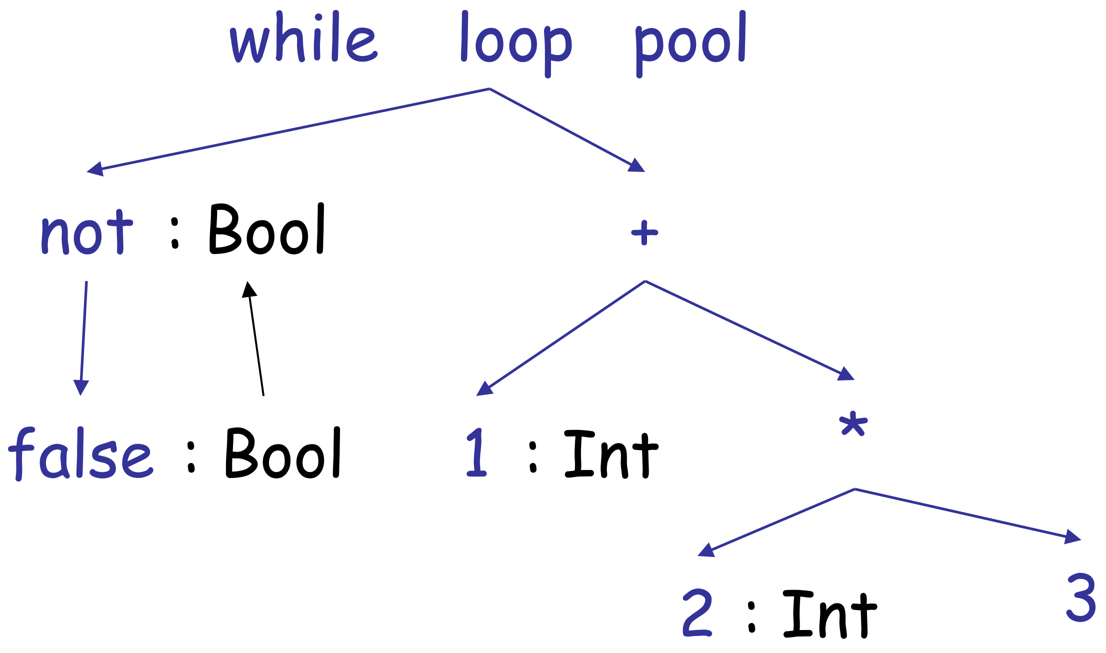
Typing: Example

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



Typing: Example

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



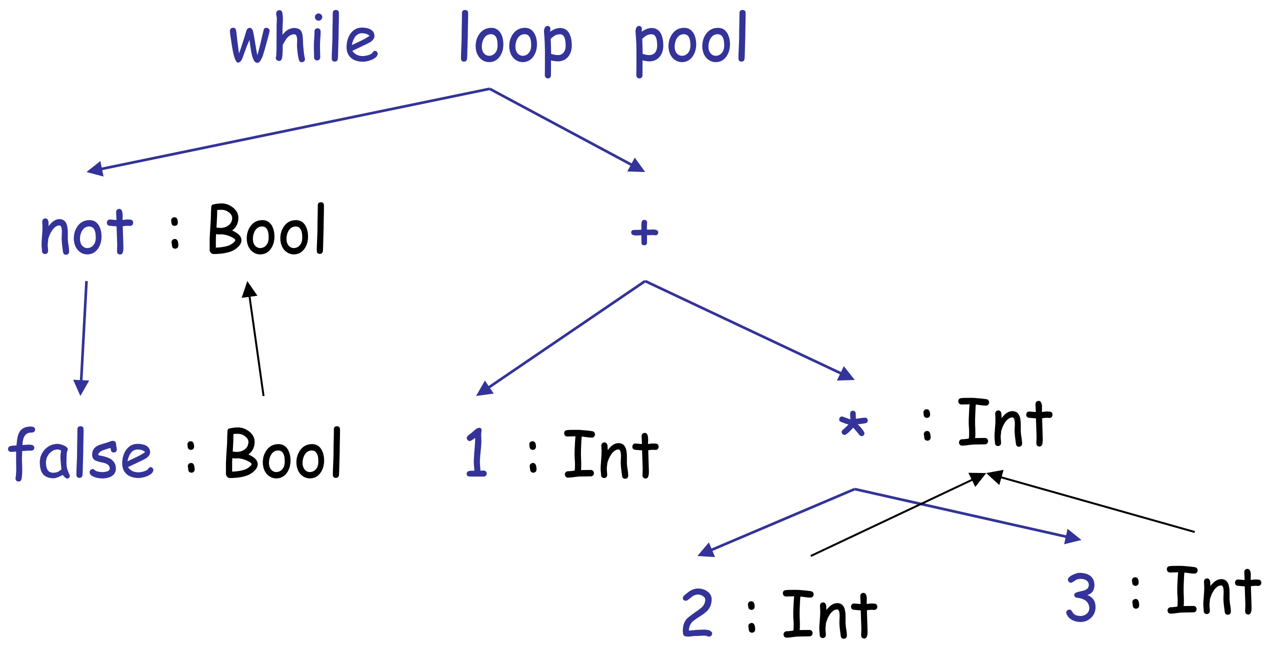
Typing: Example

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

$$\begin{tikzcd}
& \text{while} \quad \text{loop} \quad \text{pool} \arrow[dl] \arrow[dr] & & & \\
\text{not} : \text{Bool} \arrow[d, shift right] & & + \arrow[dl] \arrow[dr] & & \\
\text{false} : \text{Bool} \arrow[u, shift right] & 1 : \text{Int} & & * \arrow[dl] \arrow[dr] & \\
& & 2 : \text{Int} & & 3 : \text{Int}
\end{tikzcd}$$

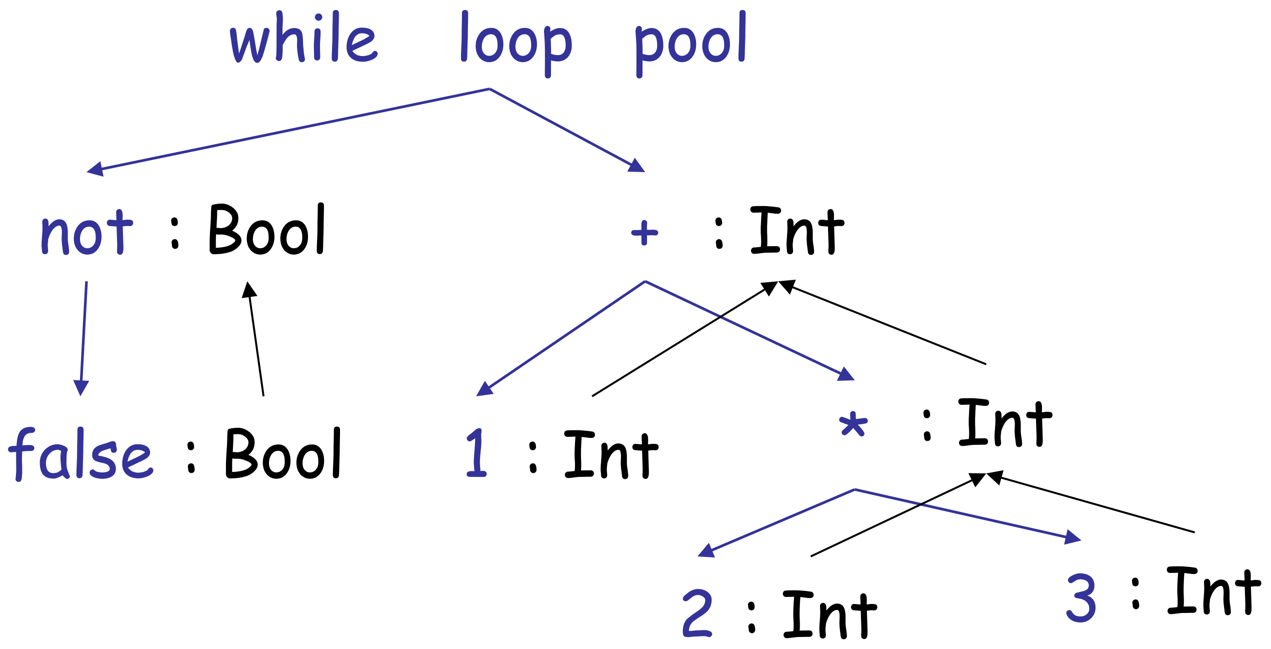
Typing: Example

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



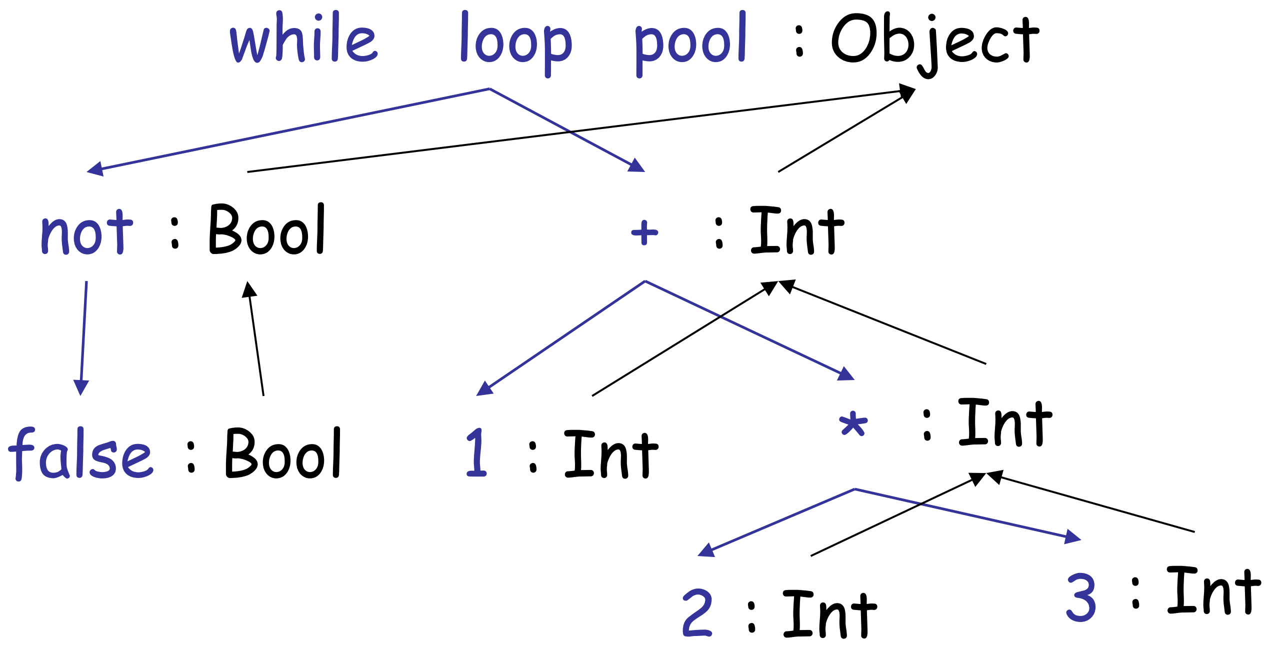
Typing: Example

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



Typing: Example

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



Typing Derivations

- The typing reasoning can be expressed as a tree:

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

- 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

A Problem

- What is the type of a variable reference?

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

- The local structural rule does *not* carry enough information to give x a type. Oh no!

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

Type Environments

Let O be a function (or mapping) 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

Modified Rules

The type environment is added to the earlier rules:

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

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

New Rules

And we can write new rules:

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

Equivalently:

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

Let

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

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

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

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

(You can write $O[x/T_0]$ on tests, etc.)

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.

Example of Typing "let"

AST

$$\begin{tikzcd}
 & \text{let } x : T_0 \text{ in} \arrow[d] & \\
 & + \arrow[dl] \arrow[dr] & \\
\text{let } y : T_1 \text{ in} \arrow[d] & & \text{let } x : T_2 \text{ in} \arrow[d] \\
E_{x,y} \arrow[d, dotted] & & F_{x,y} \\
x & &
\end{tikzcd}$$

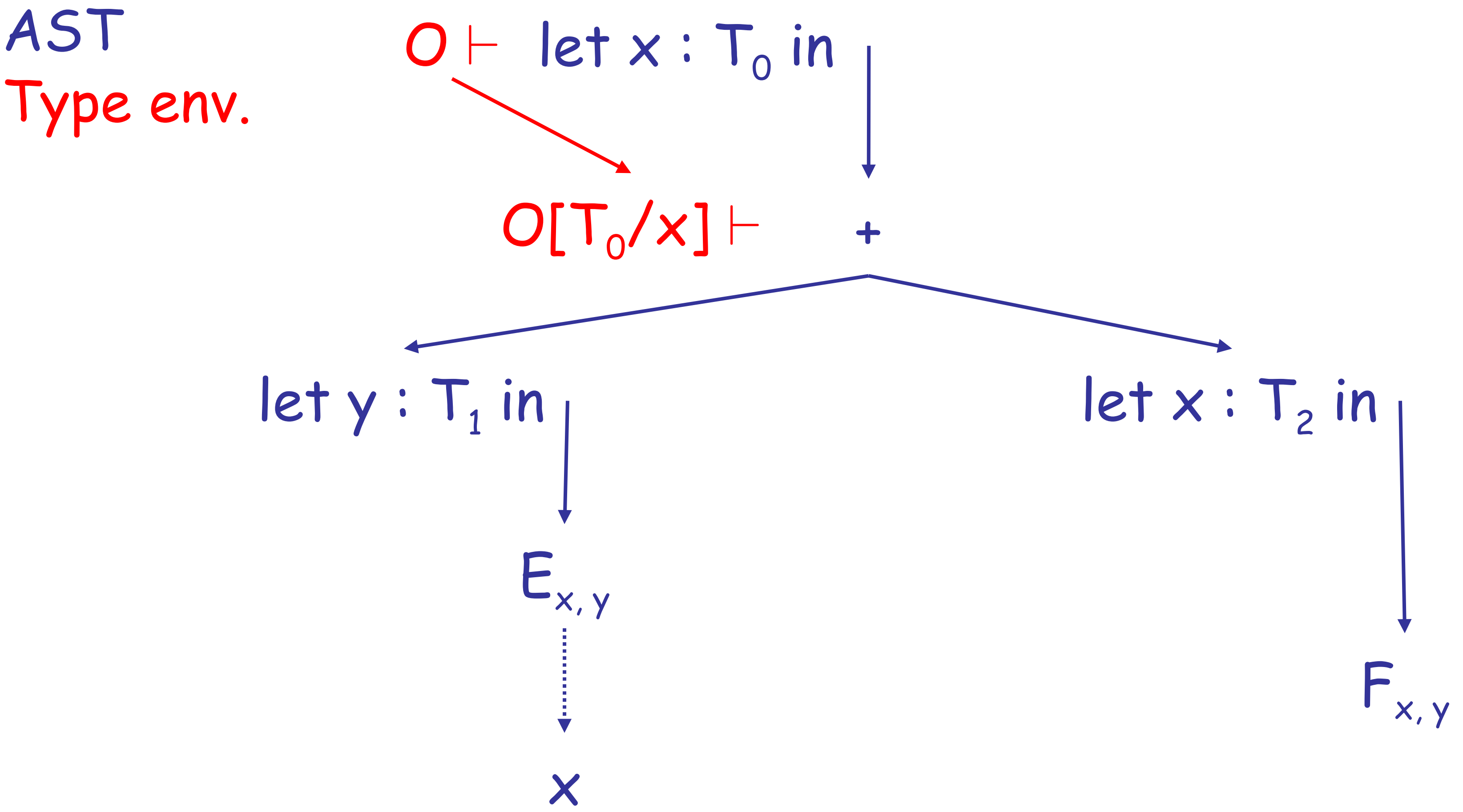
Example of Typing "let"

AST

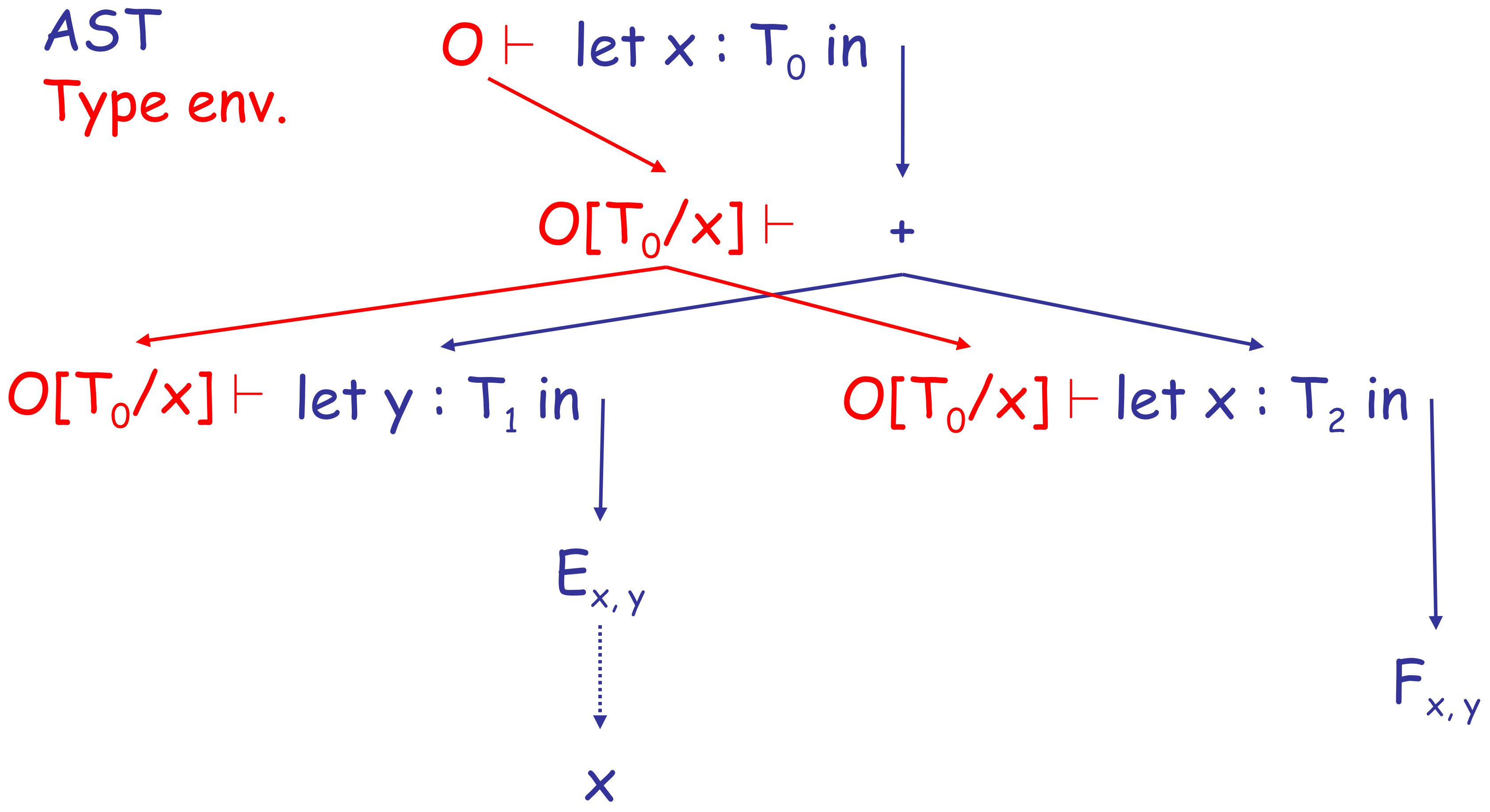
Type env.

$$\begin{tikzcd}
& O \vdash \text{let } x : T_0 \text{ in} \arrow[d] & \\
& + \arrow[dl] \arrow[dr] & \\
\text{let } y : T_1 \text{ in} \arrow[d] & & \text{let } x : T_2 \text{ in} \arrow[d] \\
E_{x,y} \arrow[d, dotted] & & F_{x,y} \\
x & &
\end{tikzcd}$$

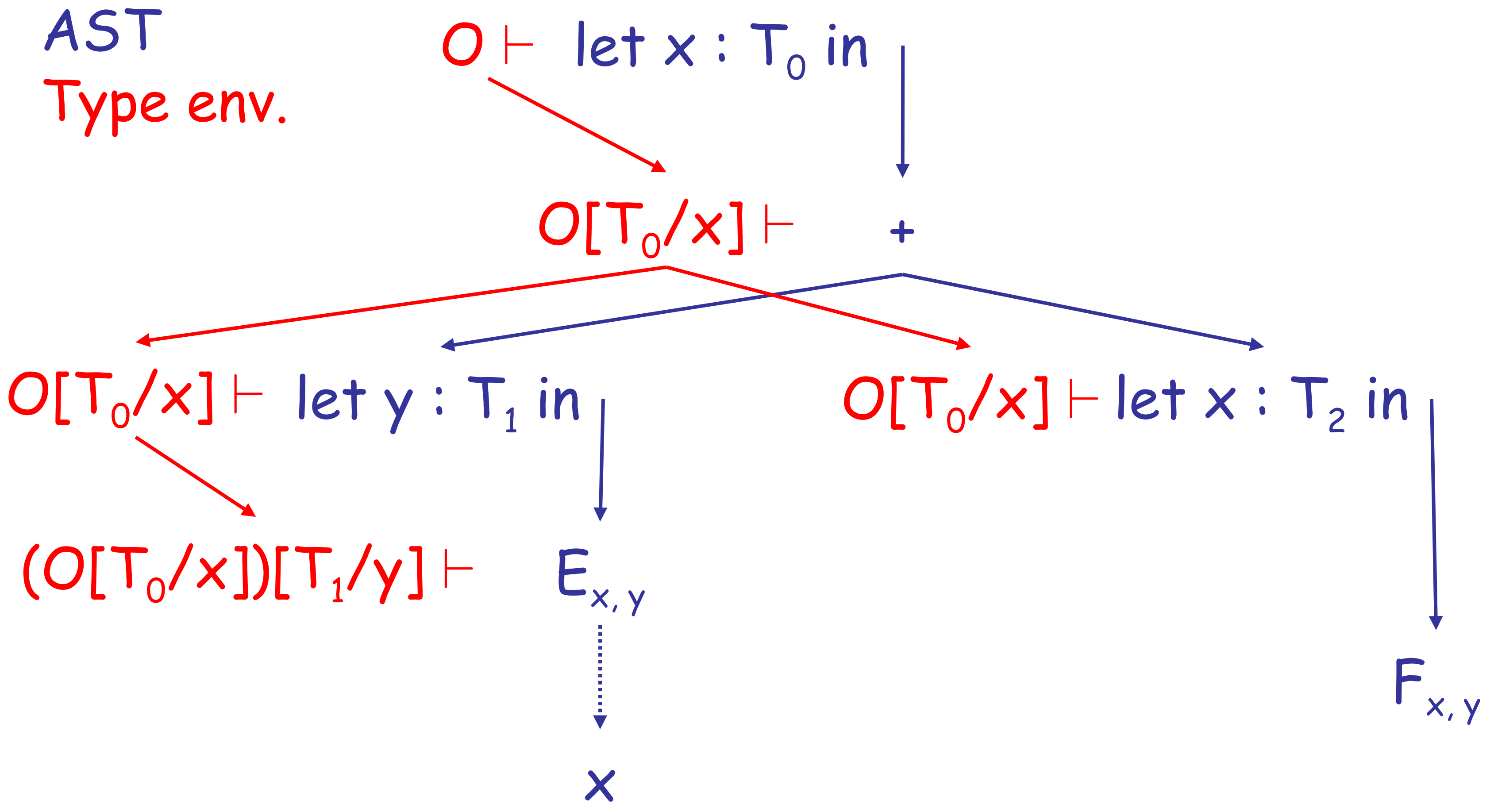
Example of Typing "let"



Example of Typing "let"



Example of Typing "let"


AST
Type env.
$O \vdash$ let $x : T_0$ in
$O[T_0/x] \vdash$ $+$
$O[T_0/x] \vdash$ let $y : T_1$ in
$O[T_0/x] \vdash$ let $x : T_2$ in
$(O[T_0/x])[T_1/y] \vdash$ $E_{x,y}$
x
$F_{x,y}$

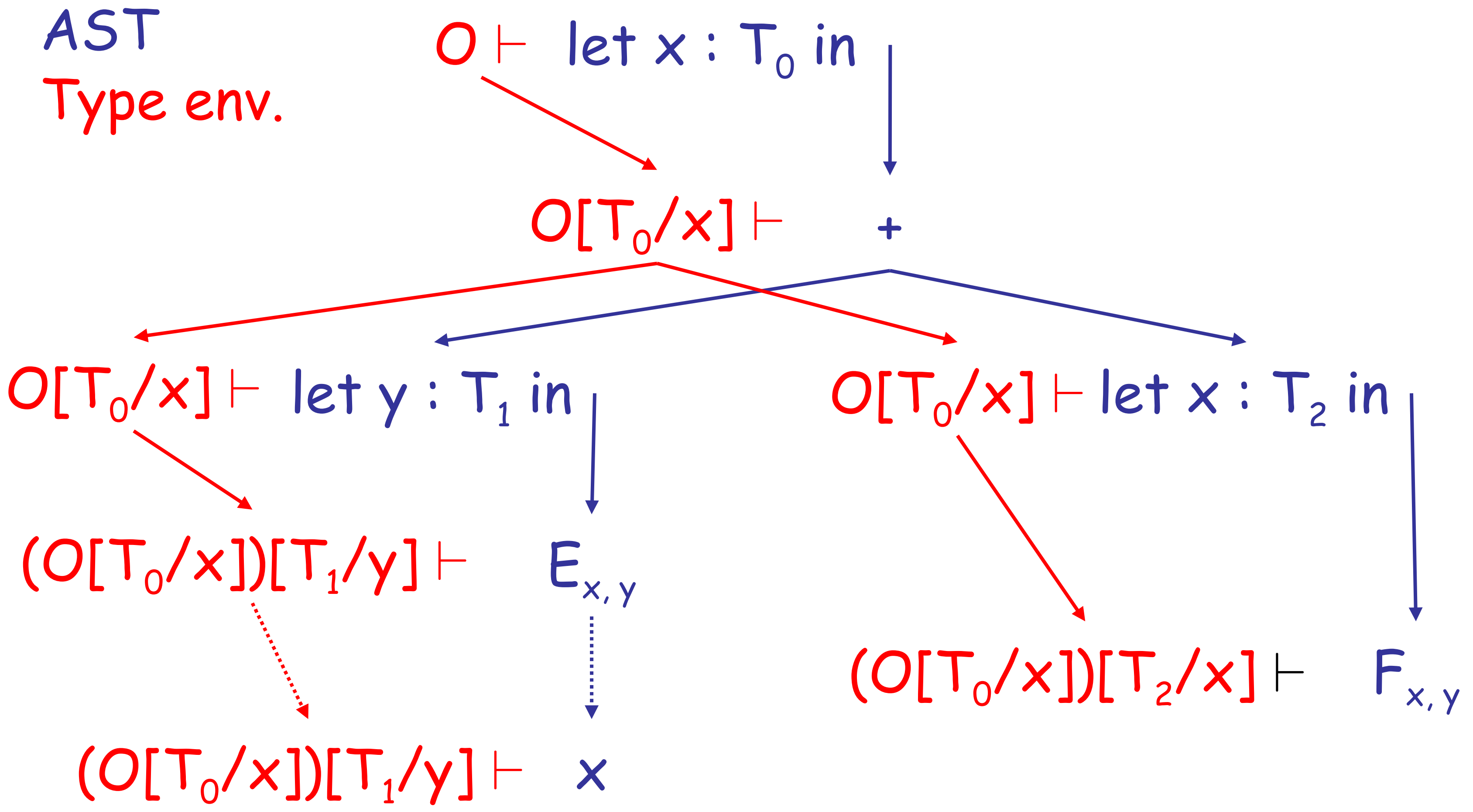
Example of Typing “let”

AST

Type env.

$$\begin{tikzcd}
 & & O \vdash \arrow[d] & \text{let } x : T_0 \text{ in} \arrow[d] & & \\
 & & O[T_0/x] \vdash \arrow[dll] \arrow[drr] & + \arrow[dll] \arrow[drr] & & \\
O[T_0/x] \vdash \arrow[d] & \text{let } y : T_1 \text{ in} \arrow[d] & & & O[T_0/x] \vdash & \text{let } x : T_2 \text{ in} \arrow[d] \\
(O[T_0/x])[T_1/y] \vdash \arrow[d, dotted] & E_{x,y} \arrow[d, dotted] & & & & F_{x,y} \\
(O[T_0/x])[T_1/y] \vdash & x & & & &
\end{tikzcd}$$

Example of Typing “let”

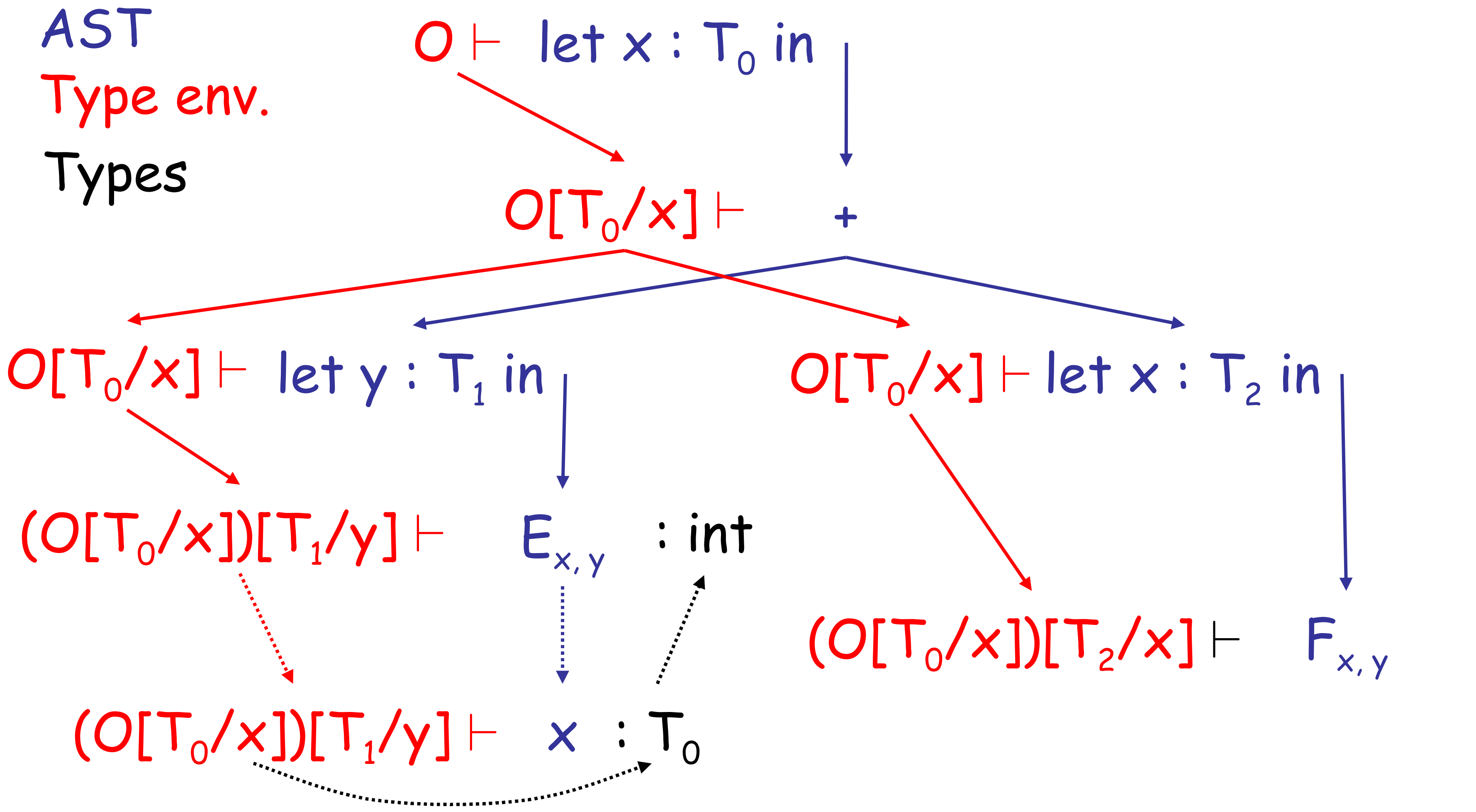

AST
Type env.
$O \vdash$ let $x : T_0$ in
$O[T_0/x] \vdash$ +
$O[T_0/x] \vdash$ let $y : T_1$ in
$O[T_0/x] \vdash$ let $x : T_2$ in
$(O[T_0/x])[T_1/y] \vdash$ $E_{x,y}$
$(O[T_0/x])[T_2/x] \vdash$ $F_{x,y}$
$(O[T_0/x])[T_1/y] \vdash$ x

Example of Typing "let"

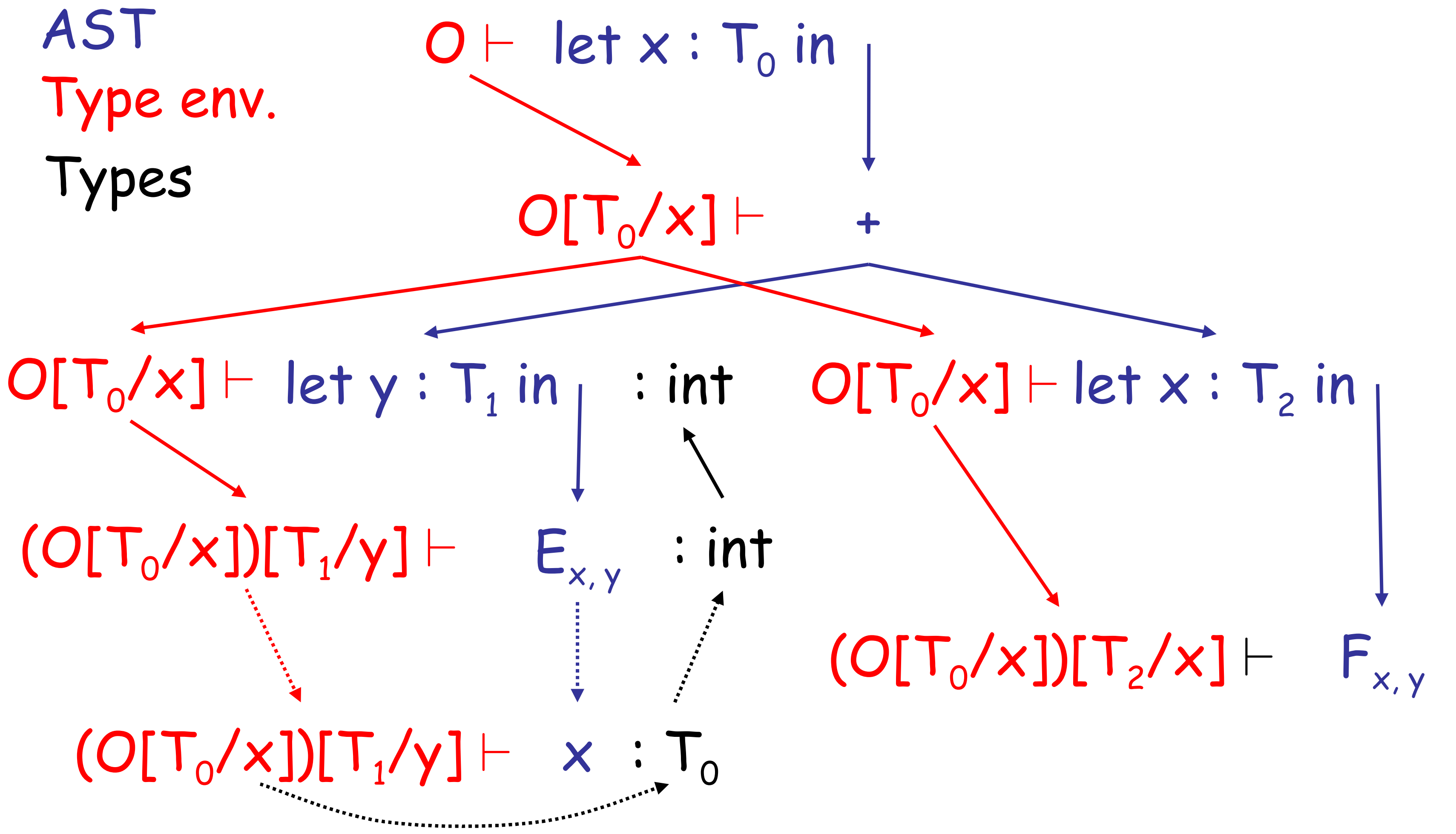
AST
Type env.
Types

$$\begin{tikzcd}
 & & O \vdash \arrow[d] & \text{let } x : T_0 \text{ in} \arrow[d] & & \\
 & & O[T_0/x] \vdash \arrow[dll] \arrow[drr] & + \arrow[dll] \arrow[drr] & & \\
O[T_0/x] \vdash \arrow[d] & \text{let } y : T_1 \text{ in} \arrow[d] & & & O[T_0/x] \vdash \arrow[d] & \text{let } x : T_2 \text{ in} \arrow[d] \\
(O[T_0/x])[T_1/y] \vdash \arrow[d, dotted] & E_{x,y} \arrow[d, dotted] & & & (O[T_0/x])[T_2/x] \vdash & F_{x,y} \\
(O[T_0/x])[T_1/y] \vdash \arrow[rr, dotted, bend right] & x & : T_0 & & &
\end{tikzcd}$$

Example of Typing “let”


AST
Type env.
Types
$O \vdash$ let x : T_0 in
$O[T_0/x] \vdash$ +
$O[T_0/x] \vdash$ let y : T_1 in
$O[T_0/x] \vdash$ let x : T_2 in
$(O[T_0/x])[T_1/y] \vdash$ $E_{x,y}$: int
$(O[T_0/x])[T_2/x] \vdash$ $F_{x,y}$
$(O[T_0/x])[T_1/y] \vdash$ x : T_0

Example of Typing "let"



AST
Type env.
Types
$O \vdash$ let x : T_0 in
$O[T_0/x] \vdash$ +
$O[T_0/x] \vdash$ let y : T_1 in
: int
$O[T_0/x] \vdash$ let x : T_2 in
$(O[T_0/x])[T_1/y] \vdash$ $E_{x,y}$: int
$(O[T_0/x])[T_2/x] \vdash$ $F_{x,y}$
$(O[T_0/x])[T_1/y] \vdash$ x : T_0

Example of Typing "let"

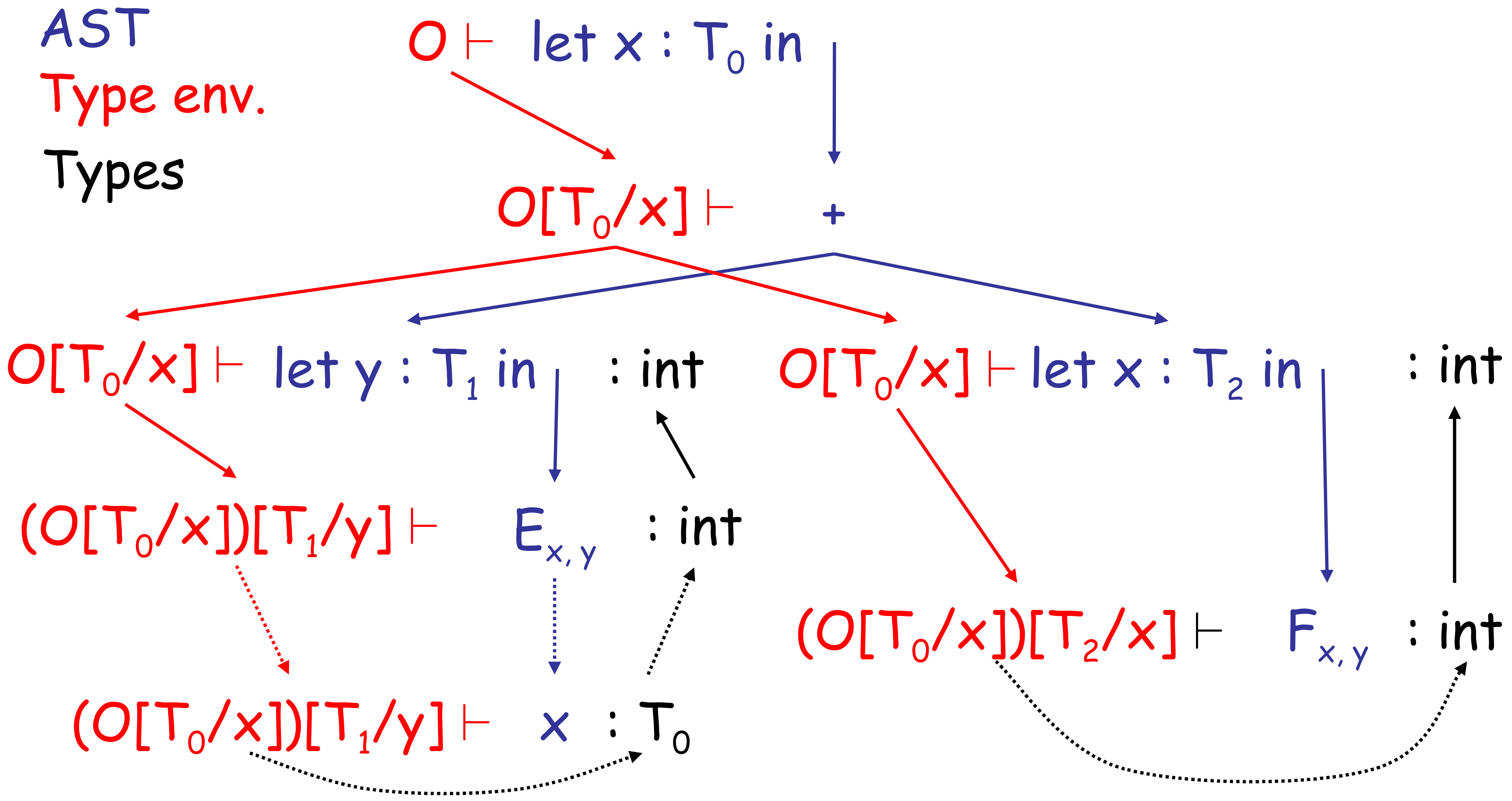
AST

Type env.

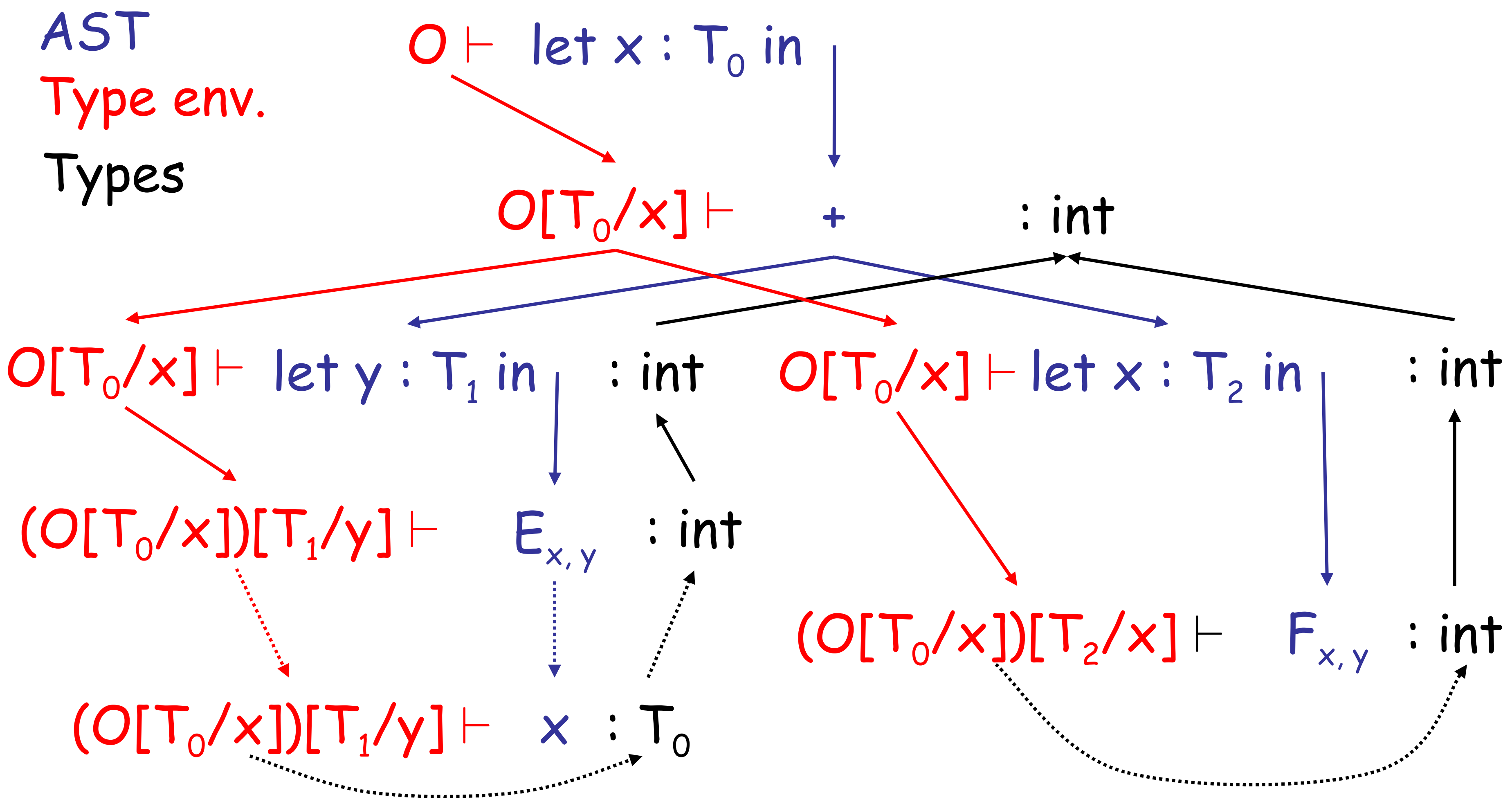
Types

$$\begin{tikzcd}
 & & & O \vdash \arrow[d] & \text{let } x : T_0 \text{ in} \arrow[d] & \\
 & & & O[T_0/x] \vdash \arrow[dlll] \arrow[d] & + \arrow[dlll] \arrow[d] & \\
O[T_0/x] \vdash \arrow[d] & \text{let } y : T_1 \text{ in} \arrow[d] & : \text{int} & O[T_0/x] \vdash \arrow[d] & \text{let } x : T_2 \text{ in} \arrow[d] & \\
(O[T_0/x])[T_1/y] \vdash \arrow[d, dotted] & E_{x,y} \arrow[d, dotted] & : \text{int} \arrow[u] & (O[T_0/x])[T_2/x] \vdash \arrow[rr, dotted, bend right] & F_{x,y} & : \text{int} \\
(O[T_0/x])[T_1/y] \vdash \arrow[rr, dotted, bend right] & x & : T_0 \arrow[u, dotted] & & &
\end{tikzcd}$$

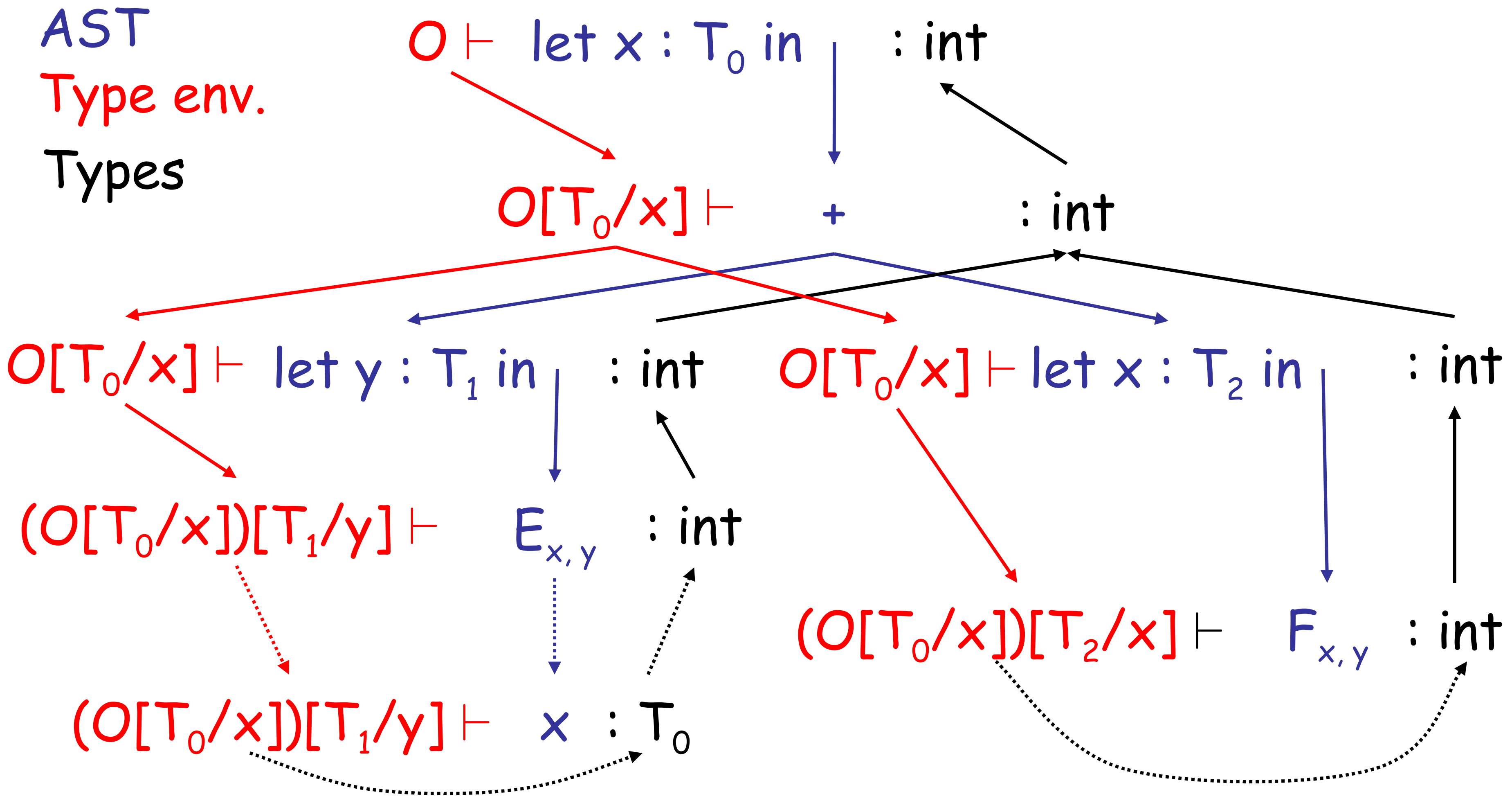
Example of Typing "let"


AST
Type env.
Types
$O \vdash$ let $x : T_0$ in
$O[T_0/x] \vdash$ +
$O[T_0/x] \vdash$ let $y : T_1$ in : int
$O[T_0/x] \vdash$ let $x : T_2$ in : int
$(O[T_0/x])[T_1/y] \vdash$ $E_{x,y}$: int
$(O[T_0/x])[T_2/x] \vdash$ $F_{x,y}$: int
$(O[T_0/x])[T_1/y] \vdash$ x : T_0

Example of Typing "let"

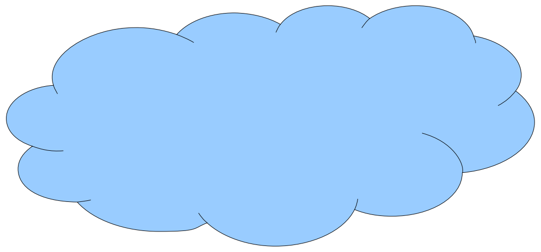

AST
Type env.
Types
O ⊢ let x : T₀ in
O[T₀/x] ⊢ + : int
O[T₀/x] ⊢ let y : T₁ in : int
O[T₀/x] ⊢ let x : T₂ in : int
(O[T₀/x])[T₁/y] ⊢ E_{x, y} : int
(O[T₀/x])[T₂/x] ⊢ F_{x, y} : int
(O[T₀/x])[T₁/y] ⊢ x : T₀

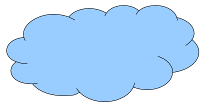
Example of Typing "let"


AST
Type env.
Types
$O \vdash$ let x : T_0 in : int
$O[T_0/x] \vdash$ + : int
$O[T_0/x] \vdash$ let y : T_1 in : int
$O[T_0/x] \vdash$ let x : T_2 in : int
$(O[T_0/x])[T_1/y] \vdash$ $E_{x,y}$: int
$(O[T_0/x])[T_2/x] \vdash$ $F_{x,y}$: int
$(O[T_0/x])[T_1/y] \vdash$ x : T_0

Practice

- Consider 1 + let x : Int in x + 2
- What would the typing derivation be?



O ⊢ 1 + let x : Int in x + 2 : 

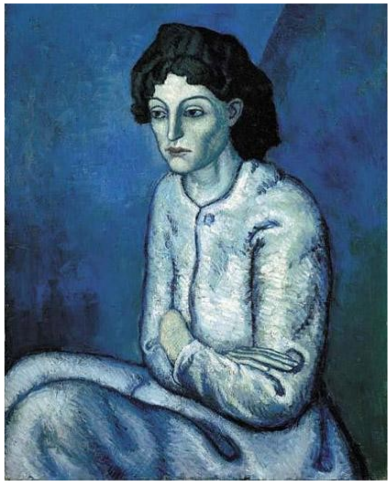
Notes

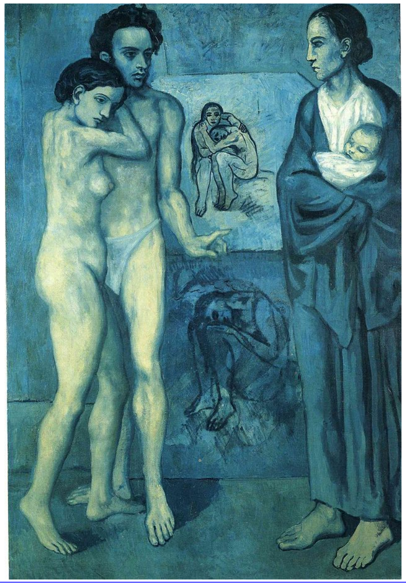
- The type environment gives types to the free identifiers in the current scope
- The **type environment** is passed down the AST from the root towards the leaves
- **Types** are computed bottom-up on the AST from the leaves toward the root

Art History Trivia

(student "memorial")

- The Período Azul refers to works produced by *this artist* between 1901 and 1904. The works place a heavy emphasis on shades of blue or blue-green, with the artist sinking into depression. While difficult to sell at the time, the works are now quite popular.





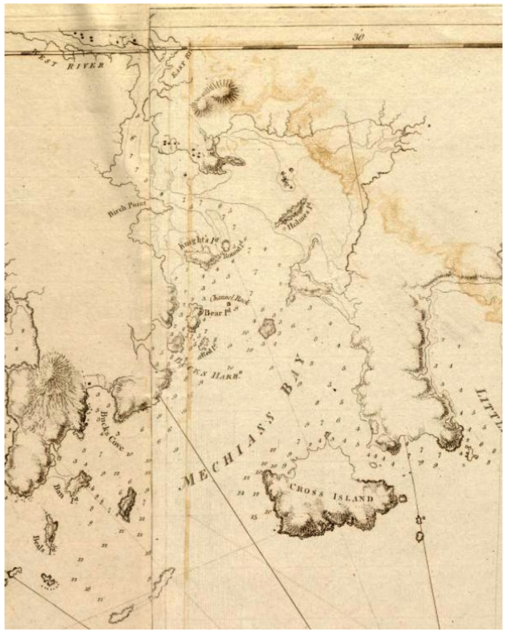
Cultural Food Trivia
(student "memorial")

- Identify the culture or ethnicity associated with the following foods:
  - Latkes (potato pancakes)
  - Poutine (fries with curds and gravy)
  - Bubble tea (tapioca, fruit, tea)
  - Sosatie (lamb or mutton on skewers with spicy sauce)

History and Geography Trivia

(student "memorial")

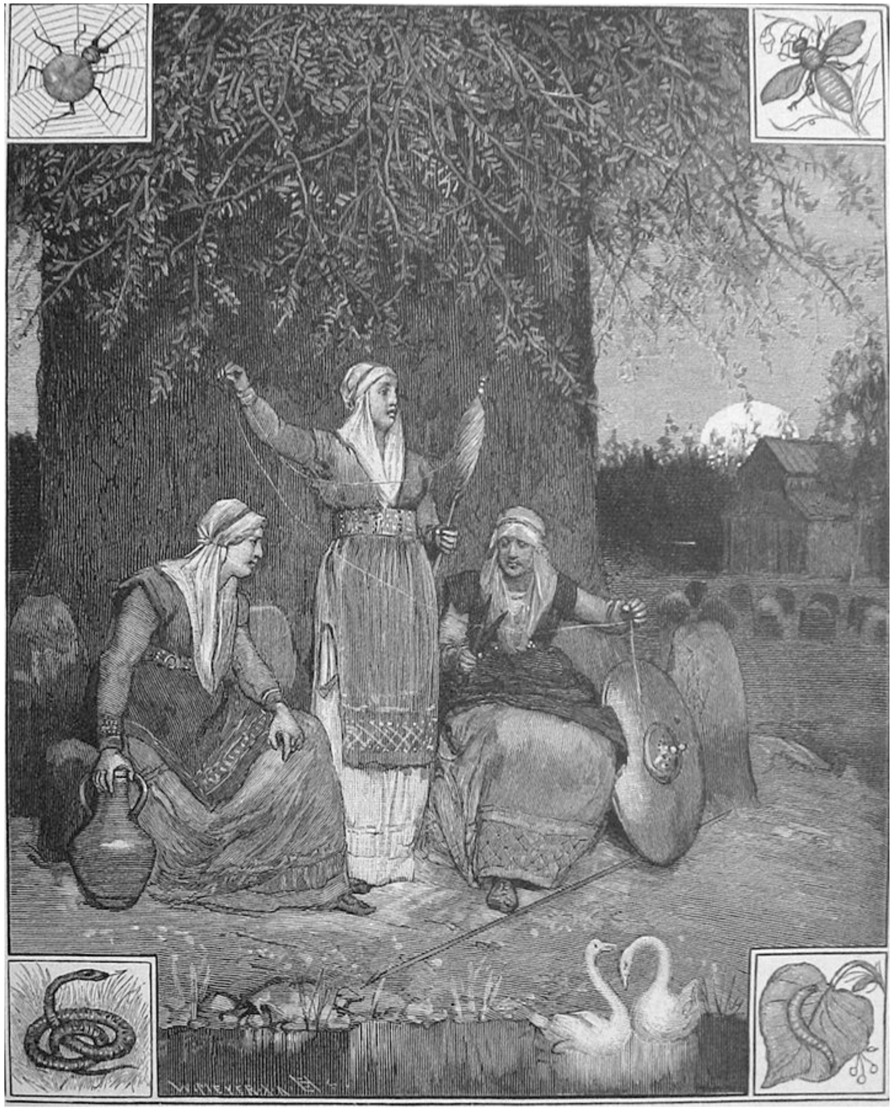
- The June 11, 1775 Battle of Machias in *this state* was the first naval battle of the American Revolutionary War. After Concord and Lexington (April 19), the Machias townspeople arrested a British loyalist and used two local ships to sail out and capture the British sloop *HMS Margaretta*. The town would see another naval battle in 1777.



Mythology Trivia

(student "memorial")

- In Norse Mythology, these three giant maidens spin the threads of fate at Yggdrasil, determining the future of each newborn.



Real-World Languages

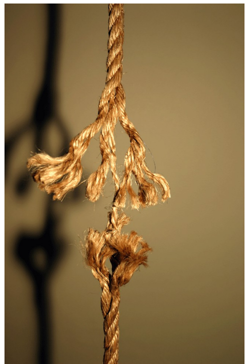
- This Indo-European language is spoken by about 100 million people (it's the most common first language in the EU). It uses an extended Latin alphabet, inflects nouns into cases (nominative, genitive, dative and accusative), and features three genders. Verb and noun inflection allow for a flexible word order. Nobel Prize winner Hermann Hesse wrote in this language.

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?



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**

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 (Better)

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

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

Type System Tug-of-War

- 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 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 (e.g., Java, Cool)

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

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 Cool type system:

$\forall$ E. dynamic_type(E) $\leq$ 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' $\leq$ 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!

Subtyping Example

- Consider the following Cool class definitions

```
Class A { a() : int { 0 }; }
Class B inherits A { b() : int { 1 }; }
```

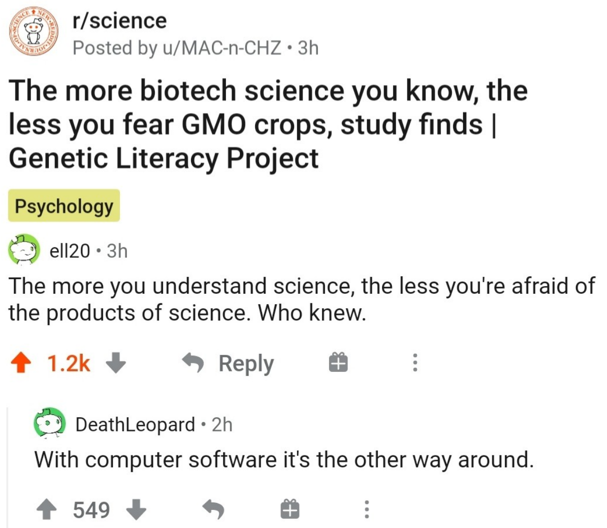
- An instance of B has methods "a" and "b"
- An instance of A has method "a"
  - A type error occurs if we try to invoke method "b" on an instance of A

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?


r/science
Posted by u/MAC-n-CHZ • 3h

The more biotech science you know, the less you fear GMO crops, study finds | Genetic Literacy Project

Psychology

ell20 • 3h

The more you understand science, the less you're afraid of the products of science. Who knew.

1.2k Reply

DeathLeopard • 2h

With computer software it's the other way around.

549

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?

Example of Wrong Let Rule (2)

- Now consider a hypothetical wrong let rule:

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

- How is it different from the correct rule?

Example of Wrong Let Rule (2)

- Now consider a hypothetical wrong let rule:

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

- 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 a hypothetical wrong let rule:

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

- How is it different from the correct rule?

THE SECRET TO HAVING A REWARDING WORK—LIFE BALANCE IS TO HAVE NO LIFE.
THEN IT'S EASY TO KEEP THINGS BALANCED BY DOING NO WORK.
SO SIMPLE, AND YET, SO GENIUS.
IT WAS HIDING IN PLAIN SIGHT.
Dilbert.com DilbertCartoonist@gmail.com
2-21-14 ©2014 Scott Adams, Inc./Dist. by Universal Uclick

Example of Wrong Let Rule (3)

- Now consider a hypothetical wrong let rule:

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

- 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?

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**

Next Time

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

Homework

- PA4c “Recommended”
  - Actually due next Tuesday