Typechecking and Static Semantics

Martin Kellogg

Today’s Agenda

- Typing Rules
- Typing Environments
- “Let” Rules
- Subtyping
- Wrong Rules

Today’s Agenda

- **Typing Rules**
- Typing Environments
- “Let” Rules
- Subtyping
- Wrong Rules

We’ll start by reviewing some of what we saw at the end of the last lecture…

English to Inference Rules

If $\mathbf{e_1}$ has type **Int** and $\mathbf{e_2}$ has type **Int**,
then $\mathbf{e_1 + e_2}$ has type **Int**

Building blocks:

- $\wedge$ is "and"
- **->** is "if-then"
- **x : T** is "x has type T"

English to Inference Rules

If $\mathbf{e_1}$ has type **Int** and $\mathbf{e_2}$ has type **Int**,
then $\mathbf{e_1 + e_2}$ has type **Int**

↓

($\mathbf{e_1}$ has type **Int** $\wedge$ $\mathbf{e_2}$ has type **Int**) ->
$\mathbf{e_1 + e_2}$ has type **Int**

Building blocks:

- $\wedge$ is "and"
- **->** is "if-then"
- **x : T** is "x has type T"

English to Inference Rules

Building blocks:

- $\land$ is "and"
- **->** is "if-then"
- **x : T** is "x has type T"

If $\mathbf{e_1}$ has type **Int** and $\mathbf{e_2}$ has type **Int**, then $\mathbf{e_1 + e_2}$ has type **Int**

↓

($\mathbf{e_1}$ has type **Int** $\land$ $\mathbf{e_2}$ has type **Int**) -> $\mathbf{e_1 + e_2}$ has type **Int**

↓

$(\mathbf{e_1} : \mathbf{Int} \land \mathbf{e_2} : \mathbf{Int}) \rightarrow \mathbf{e_1 + e_2} : \mathbf{Int}$

English to Inference Rules

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**) -> $e_1 + e_2$ has type **Int**

↓

$(e_1 : \mathbf{Int} \wedge e_2 : \mathbf{Int}) \rightarrow e_1 + e_2 : \mathbf{Int}$

Building blocks:

- $\wedge$ is "and"
- **->** is "if-then"
- **x : T** is "x has type T"

Traditional notation (same meaning!):

$$\frac{\vdash e_1 : \mathbf{Int} \qquad \vdash e_2 : \mathbf{Int}}{\vdash e_1 + e_2 : \mathbf{Int}}$$

English to Inference Rules

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** ∧ e_2 has type **Int**) ->
$e_1 + e_2$ has type **Int**

↓

(e_1 : **Int** ∧ e_2 : **Int**) -> $e_1 + e_2$: **Int**

Building blocks:

- ∧ is "and"
- **->** is "if-then"
- **x : T** is "x has type T"

Traditional notation (same meaning!):

$$\frac{\vdash e_1 : \mathrm{Int} \qquad \vdash e_2 : \mathrm{Int}}{\boxed{\vdash} e_1 + e_2 : \mathrm{Int}}$$

Pronounced "we can prove that..."

Inference Rule Examples

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

$$\frac{\textit{i is any integer constant}}{\vdash i : \mathrm{Int}}\ [\mathrm{Int}]$$

Inference Rule Examples

$$\frac{\vdash e_1 : \mathrm{Int} \qquad \vdash e_2 : \mathrm{Int}}{\vdash e_1 + e_2 : \mathrm{Int}} \;[\mathrm{Add}]$$

$$\frac{i \text{ is any integer constant}}{\vdash i : \mathrm{Int}} \;[\mathrm{Int}]$$

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

Baby's First Type Derivation

$$\dfrac{}{\vdash \mathbf{1 + 2} : \mathbf{Int}} \; \textbf{[Add]}$$

Baby's First Type Derivation

$$\dfrac{\dfrac{}{\vdash \mathbf{1} : \mathbf{Int}}\ \textbf{[Int]}}{\vdash \mathbf{1 + 2} : \mathbf{Int}}\ \textbf{[Add]}$$

Baby's First Type Derivation

$$\dfrac{\dfrac{\textit{1 is an integer constant}}{\vdash 1 : \mathrm{Int}}\,[\mathrm{Int}]}{\vdash 1 + 2 : \mathrm{Int}}\,[\mathrm{Add}]$$

Baby's First Type Derivation

$$\dfrac{\dfrac{\textit{1 is an integer constant}}{\vdash 1 : \mathrm{Int}}\,[\mathbf{Int}] \qquad \dfrac{}{\vdash 2 : \mathrm{Int}}\,[\mathbf{Int}]}{\vdash 1 + 2 : \mathrm{Int}}\,[\mathbf{Add}]$$

Baby's First Type Derivation

$$\dfrac{\dfrac{\textit{1 is an integer constant}}{\vdash \mathbf{1} : \mathbf{Int}}\;[\mathbf{Int}] \qquad \dfrac{\textit{2 is an integer constant}}{\vdash \mathbf{2} : \mathbf{Int}}\;[\mathbf{Int}]}{\vdash \mathbf{1} + \mathbf{2} : \mathbf{Int}}\;[\mathbf{Add}]$$

Baby’s First Type Derivation

“ground facts” (I will write these in *italics*)

$$\dfrac{\dfrac{\underline{\textit{1 is an integer constant}}}{\vdash 1 : \mathrm{Int}}\,[\mathrm{Int}] \qquad \dfrac{\underline{\textit{2 is an integer constant}}}{\vdash 2 : \mathrm{Int}}\,[\mathrm{Int}]}{\vdash 1 + 2 : \mathrm{Int}}\,[\mathrm{Add}]$$

Soundness

Soundness

Definition: a type system is ***sound*** if whenever $\vdash e : T$, then e evaluates to a value of type T.

Soundness

Definition: a type system is ***sound*** if whenever $\vdash e : T$, then e evaluates to a value of type T.

- Intuition: if we can prove it, then it's true!

Soundness

Definition: a type system is ***sound*** if whenever $\vdash$ e : T, then e evaluates to a value of type T.

- Intuition: if we can prove it, then it's true!
- We only want sound rules, but some sound rules are worse than others

Soundness

Definition: a type system is ***sound*** if whenever $\vdash$ e : T, then e evaluates to a value of type T.

- Intuition: if we can prove it, then it's true!
- We only want sound rules, but some sound rules are worse than others
  - e.g., consider this rule:

$$\frac{\textit{i is an integer constant}}{\vdash \mathbf{i} : \mathbf{Object}}\ \textbf{[Int-Obj]}$$

Typechecking Proofs

- Typechecking proves facts like **e : T**

Typechecking Proofs

- Typechecking proves facts like **e : T**
  - one type rule is used for each kind of expression

Typechecking Proofs

- Typechecking proves facts like **e : T**
  - one type rule is used for each kind of expression
- In the type rule used for a node e:

Typechecking Proofs

- Typechecking 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 the types of e's **subexpressions**

Typechecking Proofs

- Typechecking 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 the types of e’s **subexpressions**
  - the **conclusion** is the proof of the type of e **itself**

Typechecking Proofs

- Typechecking 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 the types of e's **subexpressions**
  - the **conclusion** is the proof of the type of e **itself**
- e.g, consider the add rule ->

$$\frac{\vdash \mathbf{e_1} : \mathbf{Int} \qquad \vdash \mathbf{e_2} : \mathbf{Int}}{\vdash \mathbf{e_1} + \mathbf{e_2} : \mathbf{Int}}\ \textbf{[Add]}$$

Typechecking Proofs

- Typechecking 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 the types of e's **subexpressions**
  - the **conclusion** is the proof of the type of e **itself**
- e.g, consider the add rule ->

subexpression proofs

$$\frac{\boxed{\vdash e_1 : \mathbf{Int}} \qquad \boxed{\vdash e_2 : \mathbf{Int}}}{\vdash e_1 + e_2 : \mathbf{Int}}\ \textbf{[Add]}$$

Typechecking Proofs

- Typechecking 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 the types of e's **subexpressions**
  - the **conclusion** is the proof of the type of e **itself**
- e.g, consider the add rule ->

subexpression proofs

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

proof of e itself

Typechecking Proofs

- Typechecking 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 the types of e's **subexpressions**
  - the **conclusion** is the proof of the type of e **itself**
- e.g, consider the add rule ->

Next, we're going to look at a collection of **examples** of type rules

subexpression proofs

$$\frac{\boxed{\vdash e_1 : \mathbf{Int}} \qquad \boxed{\vdash e_2 : \mathbf{Int}}}{\boxed{\vdash e_1 + e_2 : \mathbf{Int}}}\ \textbf{[Add]}$$

proof of e itself

Rules for Constants

Rules for Constants

$$\frac{}{\vdash \textbf{false} : \textbf{Bool}}\ [\textbf{False}]$$

Rules for Constants

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

$$\frac{}{\vdash \textbf{true} : \textbf{Bool}}\ [\text{True}]$$

Rules for Constants

$$\frac{}{\vdash \mathbf{false} : \mathbf{Bool}}\ \textbf{[False]}$$

$$\frac{}{\vdash \mathbf{true} : \mathbf{Bool}}\ \textbf{[True]}$$

$$\frac{\textit{s is any string constant}}{\vdash \mathbf{s} : \mathbf{String}}\ \textbf{[String]}$$

Rules for Constants

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

$$\frac{}{\vdash \textbf{true} : \textbf{Bool}}\ [\text{True}]$$

$$\frac{\textit{s is any string constant}}{\vdash \textbf{s} : \textbf{String}}\ [\text{String}]$$

Notation note: I'm using **bold black** for **keywords**, **bold blue** for **expressions**, and **bold red** for **types**

Rule for New

- New is a bit more complicated than constants (but not much)

Rule for New

- New is a bit more complicated than constants (but not much)
- `new T` produces an object of type `T`
  - ignore `SELF_TYPE` for now…

Rule for New

- New is a bit more complicated than constants (but not much)
- new T produces an object of type T
  - ignore SELF_TYPE for now...
- That gives us this rule:

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

Rules for Bools and Loops

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

Rules for Bools and Loops

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

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

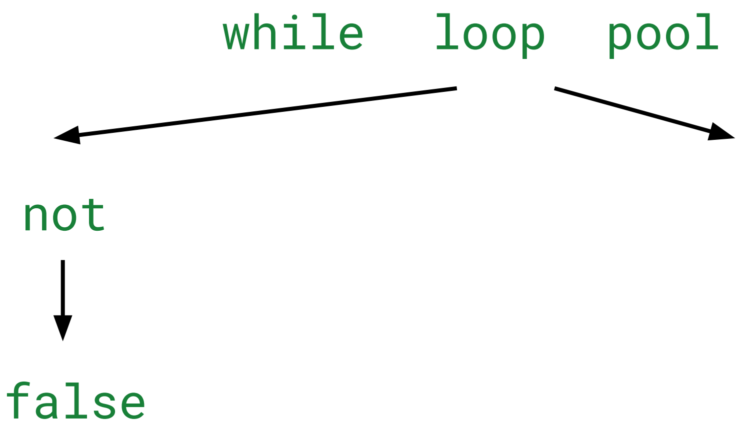
Typing Example

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

`while loop 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`

`while loop pool`

`not`

`false` : **Bool**

Typing Example

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

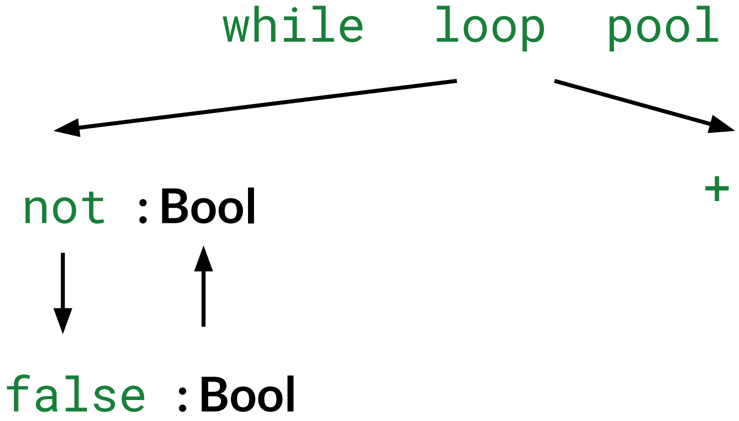
`while loop pool`

`not` : **Bool**

`false` : **Bool**

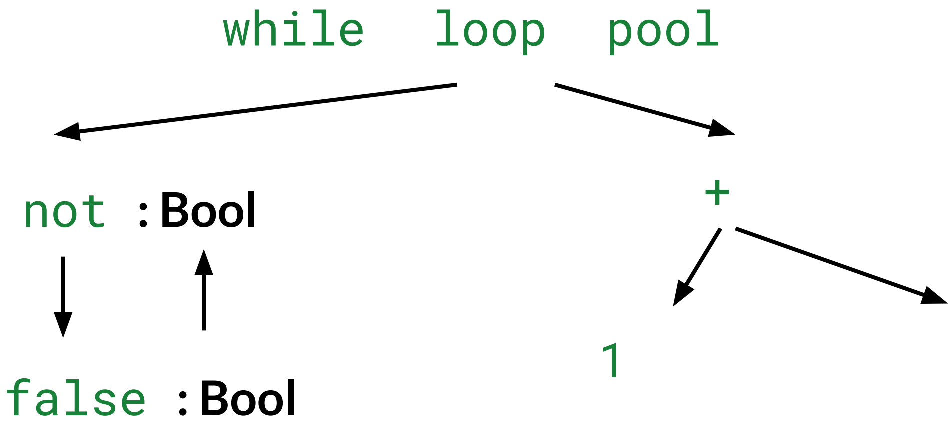
Typing Example

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



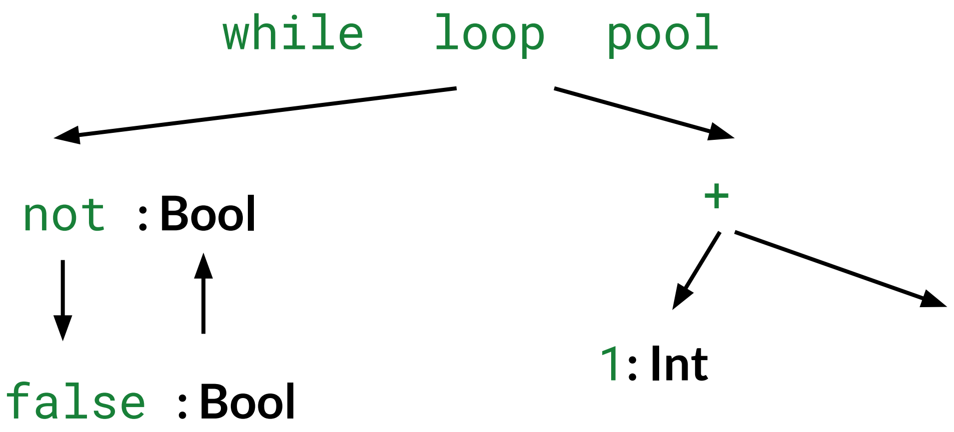
Typing Example

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



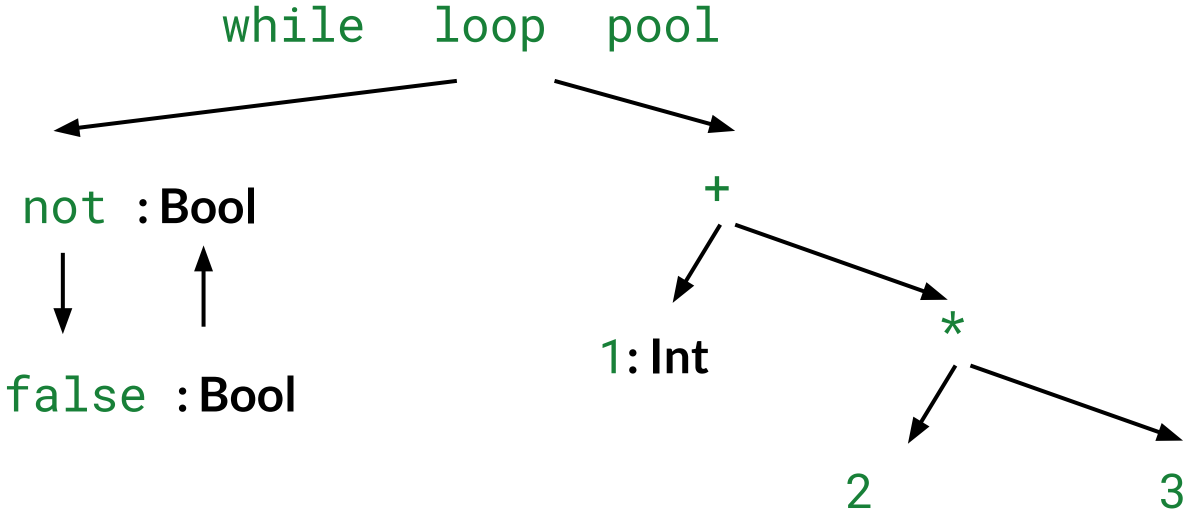
Typing Example

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



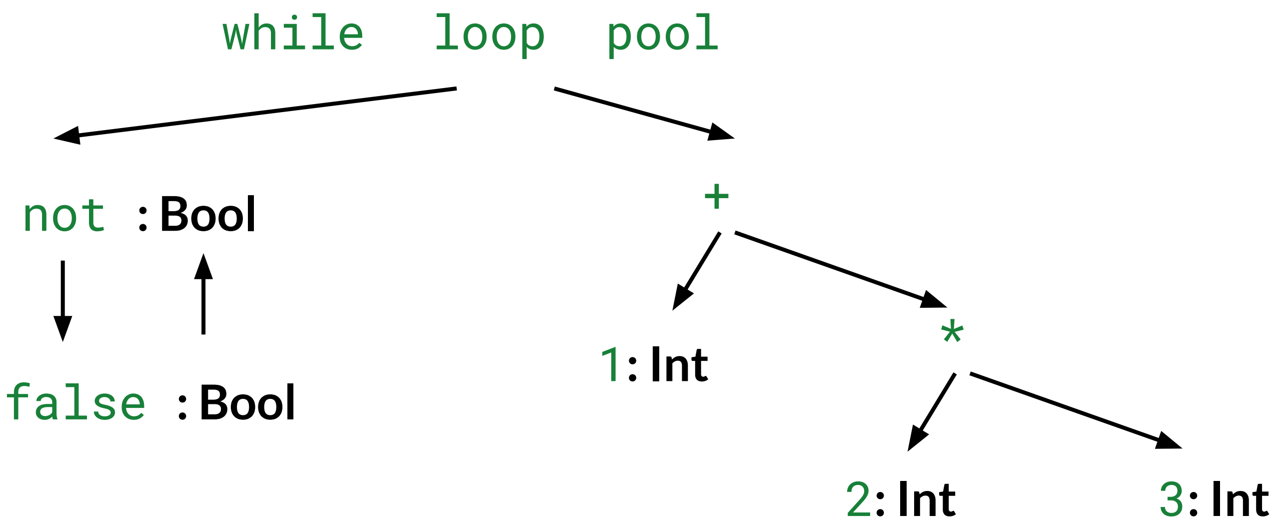
Typing Example

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



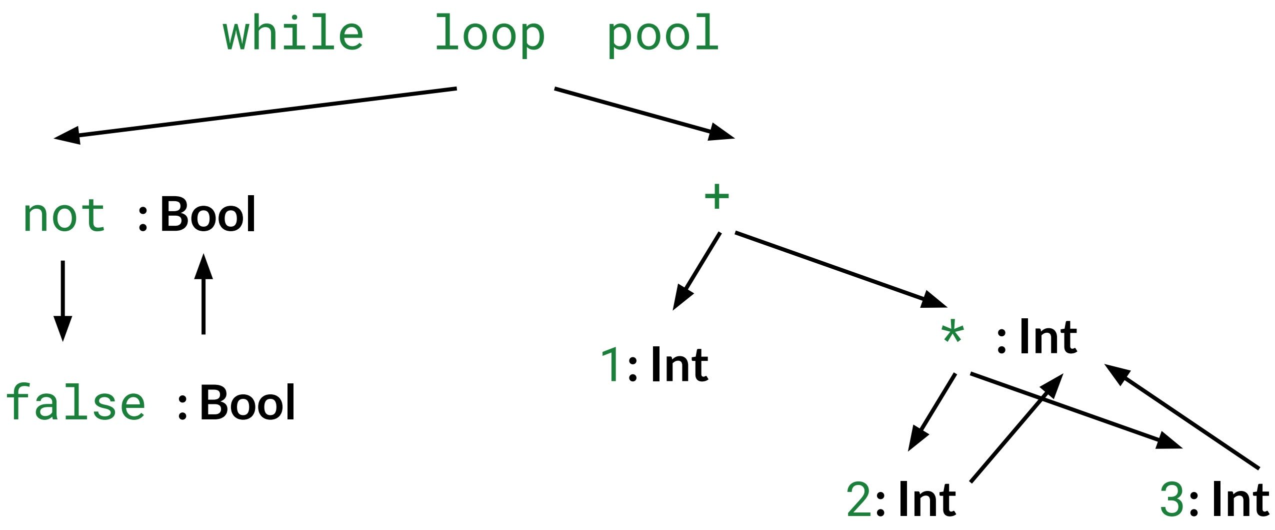
Typing Example

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



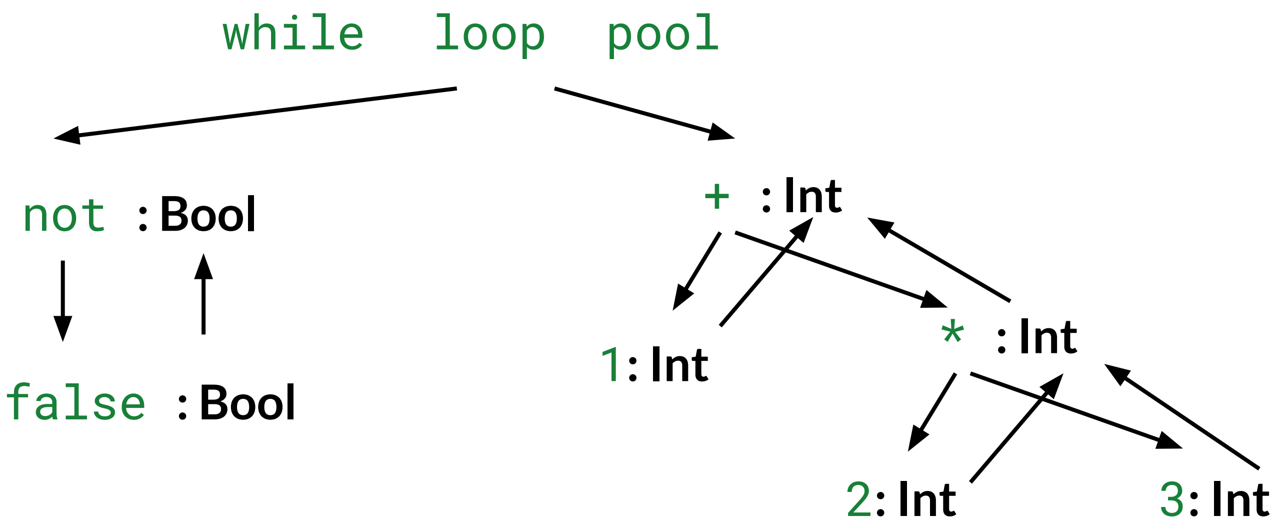
Typing Example

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



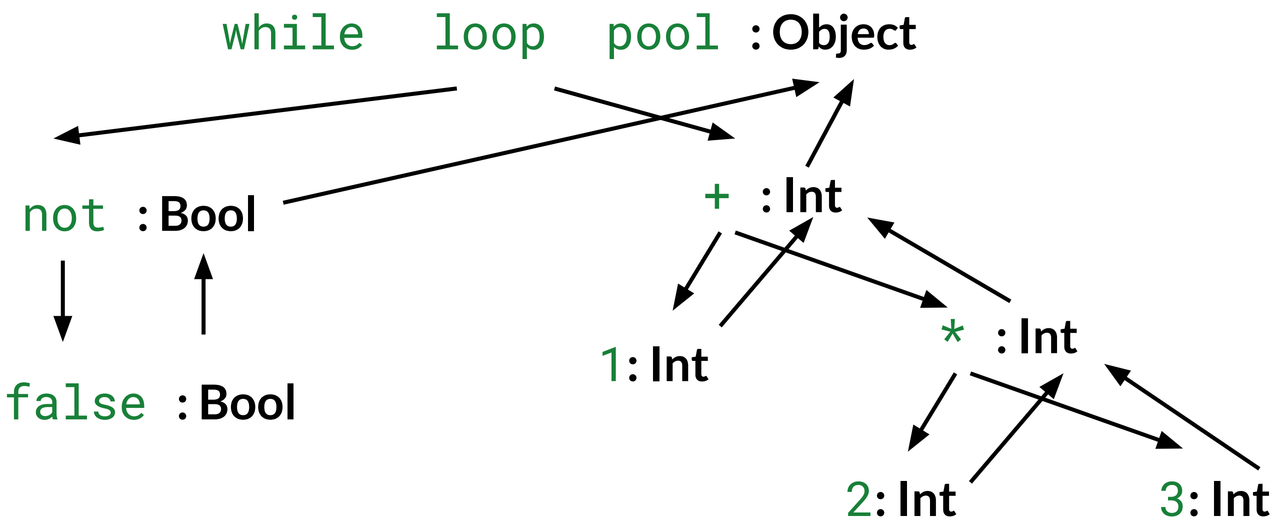
Typing Example

- 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 on the previous slide can be equivalently represented as a tree:

Typing Derivations

- The typing reasoning on the previous slide can be equivalently represented as a tree:

$$
\dfrac{
\dfrac{
\dfrac{\textit{false is a Bool}}{\vdash \texttt{false} : \textbf{Bool}}
}{\vdash \texttt{not false} : \textbf{Bool}}
\qquad
\dfrac{
\dfrac{\textit{1 is an Int}}{\vdash \texttt{1} : \textbf{Int}}
\quad
\dfrac{
\dfrac{\textit{2 is an Int}}{\vdash \texttt{2} : \textbf{Int}}
\quad
\dfrac{\textit{3 is an Int}}{\vdash \texttt{3} : \textbf{Int}}
}{\vdash \texttt{2 * 3} : \textbf{Int}}
}{\vdash \texttt{1 + 2 * 3} : \textbf{Int}}
}{\vdash \texttt{while not false loop 1 + 2 * 3 pool} : \textbf{Object}}
$$

Typing Derivations

- The typing reasoning on the previous slide can be equivalently represented as a tree:
- The **root** of the tree is the whole expression

$$\frac{\dfrac{\dfrac{\textit{false is a Bool}}{\vdash \texttt{false} : \textbf{Bool}}}{\vdash \texttt{not false} : \textbf{Bool}} \qquad \dfrac{\dfrac{\textit{1 is an Int}}{\vdash \texttt{1} : \textbf{Int}} \qquad \dfrac{\dfrac{\textit{2 is an Int}}{\vdash \texttt{2} : \textbf{Int}} \quad \dfrac{\textit{3 is an Int}}{\vdash \texttt{3} : \textbf{Int}}}{\vdash \texttt{2 * 3} : \textbf{Int}}}{\vdash \texttt{1 + 2 * 3} : \textbf{Int}}}{\vdash \texttt{while not false loop 1 + 2 * 3 pool} : \textbf{Object}}$$

Typing Derivations

- The typing reasoning on the previous slide can be equivalently represented as a tree:
- The **root** of the tree is the whole expression
- Each node is an **instance** of a typing rule

$$
\dfrac{
  \dfrac{
    \dfrac{\textit{false is a Bool}}{\vdash \texttt{false} : \textbf{Bool}}
  }{\vdash \texttt{not false} : \textbf{Bool}}
  \qquad
  \dfrac{
    \dfrac{\textit{1 is an Int}}{\vdash \texttt{1} : \textbf{Int}}
    \quad
    \dfrac{
      \dfrac{\textit{2 is an Int}}{\vdash \texttt{2} : \textbf{Int}}
      \quad
      \dfrac{\textit{3 is an Int}}{\vdash \texttt{3} : \textbf{Int}}
    }{\vdash \texttt{2 * 3} : \textbf{Int}}
  }{\vdash \texttt{1 + 2 * 3} : \textbf{Int}}
}{\vdash \texttt{while not false loop 1 + 2 * 3 pool} : \textbf{Object}}
$$

Typing Derivations

- The typing reasoning on the previous slide can be equivalently represented as a tree:
- The **root** of the tree is the whole expression
- Each node is an **instance** of a typing rule
- **Leaves** in the tree are the rules with ground facts

$$\frac{\dfrac{\dfrac{\textit{false is a Bool}}{\vdash \texttt{false} : \textbf{Bool}}}{\vdash \texttt{not false} : \textbf{Bool}} \qquad \dfrac{\dfrac{\textit{1 is an Int}}{\vdash \texttt{1} : \textbf{Int}} \quad \dfrac{\dfrac{\textit{2 is an Int}}{\vdash \texttt{2} : \textbf{Int}} \quad \dfrac{\textit{3 is an Int}}{\vdash \texttt{3} : \textbf{Int}}}{\vdash \texttt{2 * 3} : \textbf{Int}}}{\vdash \texttt{1 + 2 * 3} : \textbf{Int}}}{\vdash \texttt{while not false loop 1 + 2 * 3 pool} : \textbf{Object}}$$

A Problem: Variables

- What is the type of a variable?

A Problem: Variables

- What is the type of a variable?

$$\frac{\textit{x is an identifier}}{\vdash \mathbf{x} : \mathbf{?}}\ \textbf{[Var]}$$

A Problem: Variables

- What is the type of a variable?

$$\frac{\textit{x is an identifier}}{\vdash \mathbf{x} : \mathbf{?}}\ \textbf{[Var]}$$

- This local structural rule does **not** carry enough information to give x a type (oh no)

A Problem: Variables

- What is the type of a variable?

$$\frac{\textit{x is an identifier}}{\vdash \mathbf{x} : \mathbf{?}}\ \textbf{[Var]}$$

- This local structural rule does **not** carry enough information to give x a type (oh no)
    - All of the rules we've looked at so far have been about **constants**…

A Solution: Type Environments

- To give types to variables, our rules need to carry more information

A Solution: Type Environments

- To give types to variables, our rules need to carry more information
  - i.e., the **declared types** of variables!

A Solution: Type Environments

- To give types to variables, our rules need to carry more information
  - i.e., the **declared types** of variables!
- A ***type environment*** gives types for ***free variables***

A Solution: Type Environments

- To give types to variables, our rules need to carry more information
  - i.e., the **declared types** of variables!
- A ***type environment*** gives types for ***free variables***
  - The type environment is a **mapping** from names to types

A Solution: Type Environments

- To give types to variables, our rules need to carry more information
  - i.e., the **declared types** of variables!
- A ***type environment*** gives types for ***free variables***
  - The type environment is a **mapping** from names to types
  - A variable ***x*** is free in an expression ***e*** if ***e*** contains an occurrence of ***x*** that refers to a declaration that is **outside** of the expression

A Solution: Type Environments

- To give types to variables, our rules need to carry more information
  - i.e., the **declared types** of variables!
- A ***type environment*** gives types for ***free variables***
  - The type environment is a **mapping** from names to types
  - A variable ***x*** is free in an expression ***e*** if ***e*** contains an occurrence of ***x*** that refers to a declaration that is **outside** of the expression

Examples:

expression: “x”
free:

A Solution: Type Environments

- To give types to variables, our rules need to carry more information
  - i.e., the **declared types** of variables!
- A ***type environment*** gives types for ***free variables***
  - The type environment is a **mapping** from names to types
  - A variable ***x*** is free in an expression ***e*** if ***e*** contains an occurrence of ***x*** that refers to a declaration that is **outside** of the expression

Examples:

expression: “x”
free: “x”

e: “let x : Int in x + y”
free:

A Solution: Type Environments

- To give types to variables, our rules need to carry more information
  - i.e., the **declared types** of variables!
- A ***type environment*** gives types for ***free variables***
  - The type environment is a **mapping** from names to types
  - A variable ***x*** is free in an expression ***e*** if ***e*** contains an occurrence of ***x*** that refers to a declaration that is **outside** of the expression

Examples:

expression: “x”
free: “x”

e: “let x : Int in x + y”
free: “y”

e: “x + let x : Int in x + y”
free:

A Solution: Type Environments

- To give types to variables, our rules need to carry more information
    - i.e., the **declared types** of variables!
- A ***type environment*** gives types for ***free variables***
    - The type environment is a **mapping** from names to types
    - A variable ***x*** is free in an expression ***e*** if ***e*** contains an occurrence of ***x*** that refers to a declaration that is **outside** of the expression

Examples:

expression: “x”
free: “x”

e: “let x : Int in x + y”
free: “y”

e: “x + let x : Int in x + y”
free: “x”, “y”

A Solution: Type Environments

- Let Γ be a **function** (equivalently, a mapping) from identifiers to types

A Solution: Type Environments

- Let Γ be a **function** (equivalently, a mapping) from identifiers to types
- Then the sentence:

$$\Gamma \vdash e : T$$

is read: "Under the **assumption** that each free variable x in e has the type given by Γ(x), then it is provable that the expression e has type T.

A Solution: Type Environments

- Let Γ be a **function** (equivalently, a mapping) from identifiers to types
- Then the sentence:

$$\Gamma \vdash e : T$$

is read: "Under the **assumption** that each free variable x in e has the type given by Γ(x), then it is provable that the expression e has type T.

> "Γ" is a **capital gamma** (the third letter of the Greek alphabet). We use Γ for type environments by convention.

Modified Rules

Modified Rules

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

$$\frac{i \text{ is any integer constant}}{\Gamma \vdash i : \text{Int}}\ [\text{Int}]$$

New Rules

New Rules

$$\frac{x \textit{ is an identifier} \qquad \Gamma(\mathbf{x}) = \mathbf{T}}{\Gamma \vdash \mathbf{x} : \mathbf{T}} \; \textbf{[Var]}$$

New Rules

$$\frac{\textit{x is an identifier} \qquad \Gamma(\mathbf{x}) = \mathbf{T}}{\Gamma \vdash \mathbf{x} : \mathbf{T}} \textbf{[Var]}$$

- We're almost ready for today's main event: the **let rule**

New Rules

$$\frac{\textit{x is an identifier} \qquad \Gamma(\mathbf{x}) = \mathbf{T}}{\Gamma \vdash \mathbf{x} : \mathbf{T}} \text{ [Var]}$$

- We're almost ready for today's main event: the **let rule**
- First, we need one more piece of notation: $\Gamma[\mathbf{T}/\mathbf{x}]$

New Rules

$$\frac{x \text{ is an identifier} \qquad \Gamma(\mathbf{x}) = \mathbf{T}}{\Gamma \vdash \mathbf{x} : \mathbf{T}} \textbf{[Var]}$$

- We're almost ready for today's main event: the **let rule**
- First, we need one more piece of notation: $\Gamma[\mathbf{T}/\mathbf{x}]$
- $\Gamma[\mathbf{T}/\mathbf{x}]$ means "Γ modified to map $\mathbf{x}$ to $\mathbf{T}$, and behaving as Γ on all other arguments"

New Rules

$$\frac{x \text{ is an identifier} \qquad \Gamma(\mathbf{x}) = \mathbf{T}}{\Gamma \vdash \mathbf{x} : \mathbf{T}} \; \textbf{[Var]}$$

- We're almost ready for today's main event: the **let rule**
- First, we need one more piece of notation: $\Gamma[\mathbf{T}/\mathbf{x}]$
- $\Gamma[\mathbf{T}/\mathbf{x}]$ means "Γ modified to map **x** to **T**, and behaving as Γ on all other arguments"
- More formally:

New Rules

$$\frac{\textit{x is an identifier} \qquad \Gamma(\mathbf{x}) = \mathbf{T}}{\Gamma \vdash \mathbf{x} : \mathbf{T}} \ \textbf{[Var]}$$

- We're almost ready for today's main event: the **let rule**
- First, we need one more piece of notation: $\Gamma[\mathbf{T}/\mathbf{x}]$
- $\Gamma[\mathbf{T}/\mathbf{x}]$ means "Γ modified to map $\mathbf{x}$ to $\mathbf{T}$, and behaving as Γ on all other arguments"
- More formally: $\Gamma[\mathbf{T}/\mathbf{x}](\mathbf{x}) = \mathbf{T}$

New Rules

$$\frac{\textit{x is an identifier} \qquad \Gamma(\mathbf{x}) = \mathbf{T}}{\Gamma \vdash \mathbf{x} : \mathbf{T}} \textbf{[Var]}$$

- We're almost ready for today's main event: the **let rule**
- First, we need one more piece of notation: $\Gamma[\mathbf{T}/\mathbf{x}]$
- $\Gamma[\mathbf{T}/\mathbf{x}]$ means "Γ modified to map $\mathbf{x}$ to $\mathbf{T}$, and behaving as Γ on all other arguments"
- More formally: $\Gamma[\mathbf{T}/\mathbf{x}](\mathbf{x}) = \mathbf{T} \land \Gamma[\mathbf{T}/\mathbf{x}](\mathbf{y}) = \Gamma(\mathbf{y}) \ \forall \ \mathbf{y} \neq \mathbf{x}$

New Rules

$$\frac{x \text{ is an identifier} \qquad \Gamma(\mathbf{x}) = \mathbf{T}}{\Gamma \vdash \mathbf{x} : \mathbf{T}} \; \textbf{[Var]}$$

You can write $\Gamma[\mathbf{x}/\mathbf{T}]$ on tests, etc., without penalty.

- We're almost ready for today's main event: the **let rule**
- First, we need one more piece of notation: $\Gamma[\mathbf{T}/\mathbf{x}]$
- $\Gamma[\mathbf{T}/\mathbf{x}]$ means "Γ modified to map $\mathbf{x}$ to $\mathbf{T}$, and behaving as Γ on all other arguments"
- More formally: $\Gamma[\mathbf{T}/\mathbf{x}](\mathbf{x}) = \mathbf{T} \land \Gamma[\mathbf{T}/\mathbf{x}](\mathbf{y}) = \Gamma(\mathbf{y}) \;\forall\; \mathbf{y} \neq \mathbf{x}$

Notation: $\Gamma[T/x](x) = T \wedge \Gamma[T/x](y) = \Gamma(y)\ \forall\ y \neq x$

New Rules: Let

Notation: $\Gamma[T/x](x) = T \land \Gamma[T/x](y) = \Gamma(y)\ \forall\ y \neq x$

New Rules: Let

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

Explanation:

Notation: $\Gamma[T/x](x) = T \wedge \Gamma[T/x](y) = \Gamma(y)\ \forall\ y \neq x$

New Rules: Let

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

Explanation:

- if, after replacing x's old type with T_0, we can prove that e_1 has type T_1...

Notation: $\Gamma[T/x](x) = T \wedge \Gamma[T/x](y) = \Gamma(y)\ \forall\ y \neq x$

New Rules: Let

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

Explanation:

- if, after replacing x's old type with T_0, we can prove that e_1 has type T_1...
- ...then the **let** expression is well-typed by T_1

Let Example

- Consider this Cool expression:

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

  (where $E_{x,y}$ and $F_{x,y}$ are some Cool expressions that contain occurrences of “x” and “y”)

Let Example

- Consider this Cool expression:

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

  (where $\mathbf{E}_{x,y}$ and $\mathbf{F}_{x,y}$ are some Cool expressions that contain occurrences of “x” and “y”)

- Consider that the **scope**:

Let Example

- Consider this Cool expression:

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

  (where $\mathbf{E_{x,y}}$ and $\mathbf{F_{x,y}}$ are some Cool expressions that contain occurrences of “x” and “y”)
- Consider that the **scope**:
  - of “y” is $\mathbf{E_{x,y}}$

Let Example

- Consider this Cool expression:

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

  (where $\mathbf{E_{x,y}}$ and $\mathbf{F_{x,y}}$ are some Cool expressions that contain occurrences of “x” and “y”)

- Consider that the **scope**:
  - of “y” is $\mathbf{E_{x,y}}$
  - of the outer “x” is $\mathbf{E_{x,y}}$

Let Example

- Consider this Cool expression:

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

  (where $\mathbf{E}_{x,y}$ and $\mathbf{F}_{x,y}$ are some Cool expressions that contain occurrences of “x” and “y”)

- Consider that the **scope**:
  - of “y” is $\mathbf{E}_{x,y}$
  - of the outer “x” is $\mathbf{E}_{x,y}$
  - of the inner “x” is $\mathbf{F}_{x,y}$

Let Example

- Consider this Cool expression:

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

  (where $\mathbf{E}_{x,y}$ and $\mathbf{F}_{x,y}$ are some Cool expressions that contain occurrences of "x" and "y")
- Consider that the **scope**:
  - of "y" is $\mathbf{E}_{x,y}$
  - of the outer "x" is $\mathbf{E}_{x,y}$
  - of the inner "x" is $\mathbf{F}_{x,y}$
- The type rule **precisely captures** this scoping!

Let Example

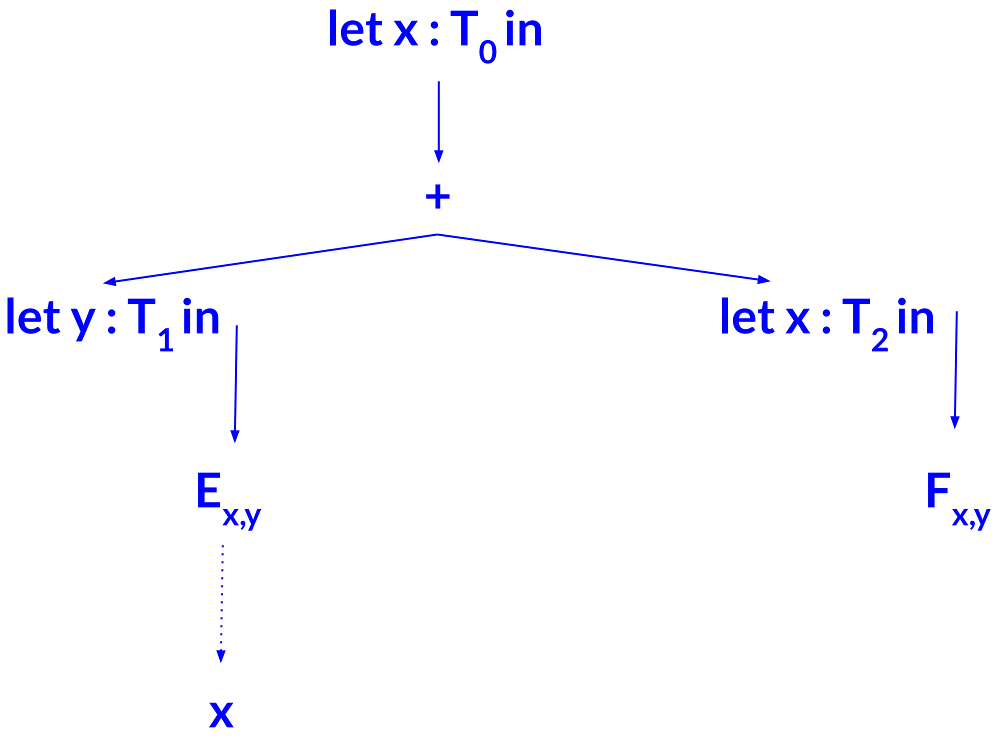
- Consider this Cool expression:

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

  (where $\mathbf{E_{x,y}}$ and $\mathbf{F_{x,y}}$ are some Cool expressions that contain occurrences of "x" and "y")
- Consider that the **scope**:
  - of "y" is $\mathbf{E_{x,y}}$
  - of the outer "x" is $\mathbf{E_{x,y}}$
  - of the inner "x" is $\mathbf{F_{x,y}}$
- The type rule **precisely captures** this scoping!
  - "Γ[T/x]"-like replacements **exactly match** the scoping we'd expect!

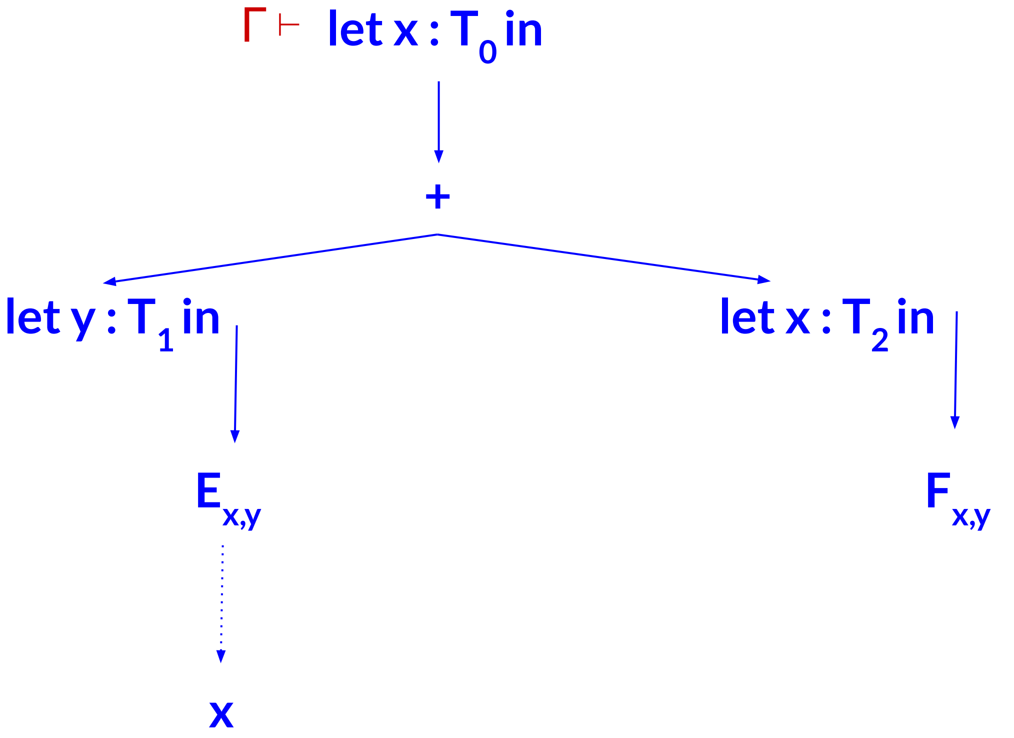
Example of Typing Let

AST



Example of Typing Let

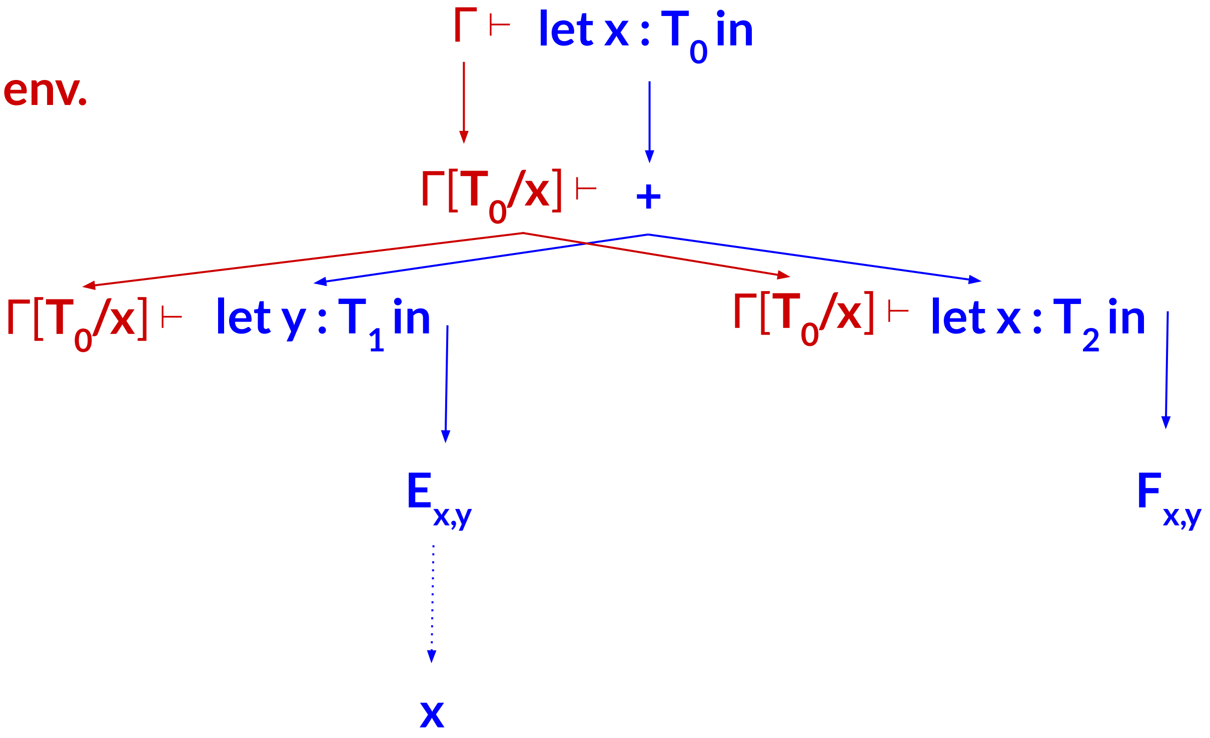
AST
Type env.



Example of Typing Let

AST

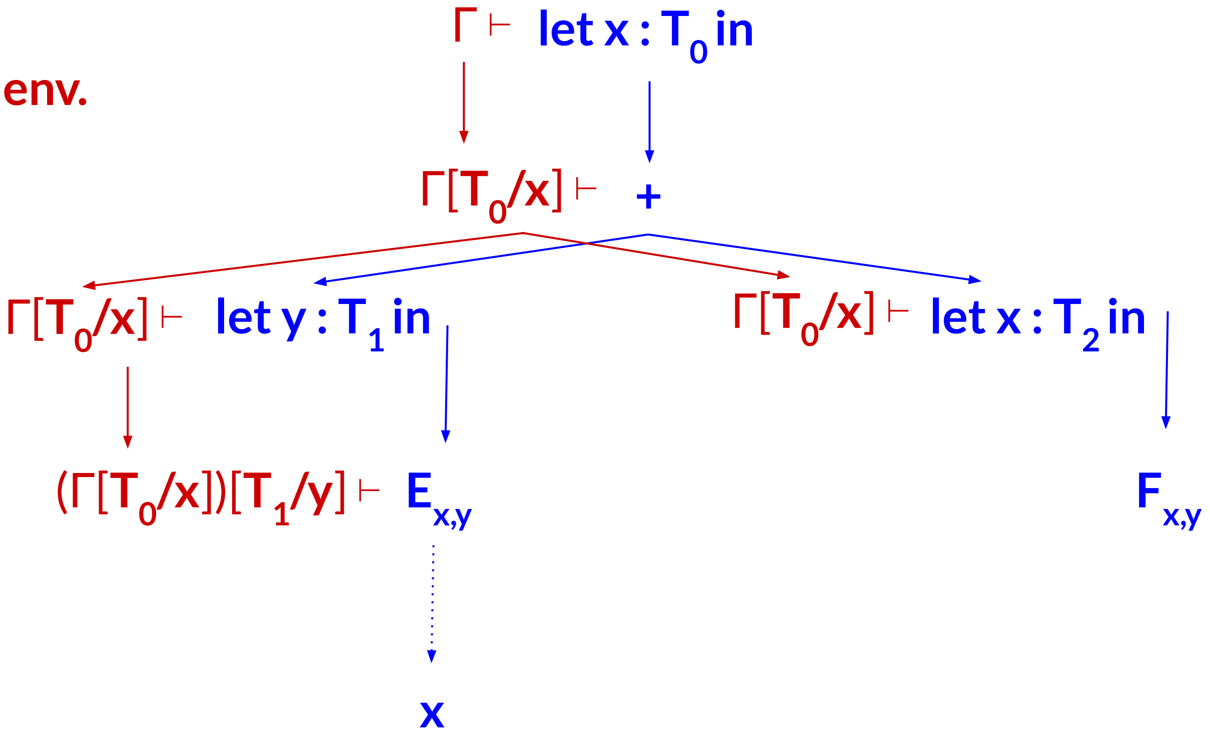
Type env.



Example of Typing Let

AST

Type env.



Example of Typing Let

AST

Type env.

$\Gamma \vdash$ **let** $x : T_0$ **in**

$\Gamma[T_0/x] \vdash +$

$\Gamma[T_0/x] \vdash$ **let** $y : T_1$ **in**

$\Gamma[T_0/x] \vdash$ **let** $x : T_2$ **in**

$(\Gamma[T_0/x])[T_1/y] \vdash E_{x,y}$

$F_{x,y}$

$(\Gamma[T_0/x])[T_1/y] \vdash x$

Example of Typing Let

AST
Type env.

$\Gamma \vdash \textbf{let } x : T_0 \textbf{ in}$

$\Gamma[T_0/x] \vdash +$

$\Gamma[T_0/x] \vdash \textbf{let } y : T_1 \textbf{ in}$

$\Gamma[T_0/x] \vdash \textbf{let } x : T_2 \textbf{ in}$

$(\Gamma[T_0/x])[T_1/y] \vdash E_{x,y}$

$(\Gamma[T_0/x])[T_2/x] \vdash F_{x,y}$

$(\Gamma[T_0/x])[T_1/y] \vdash x$

Example of Typing Let

AST
Type env.
Types

$\Gamma \vdash$ let $x : T_0$ in

$\Gamma[T_0/x] \vdash +$

$\Gamma[T_0/x] \vdash$ let $y : T_1$ in

$\Gamma[T_0/x] \vdash$ let $x : T_2$ in

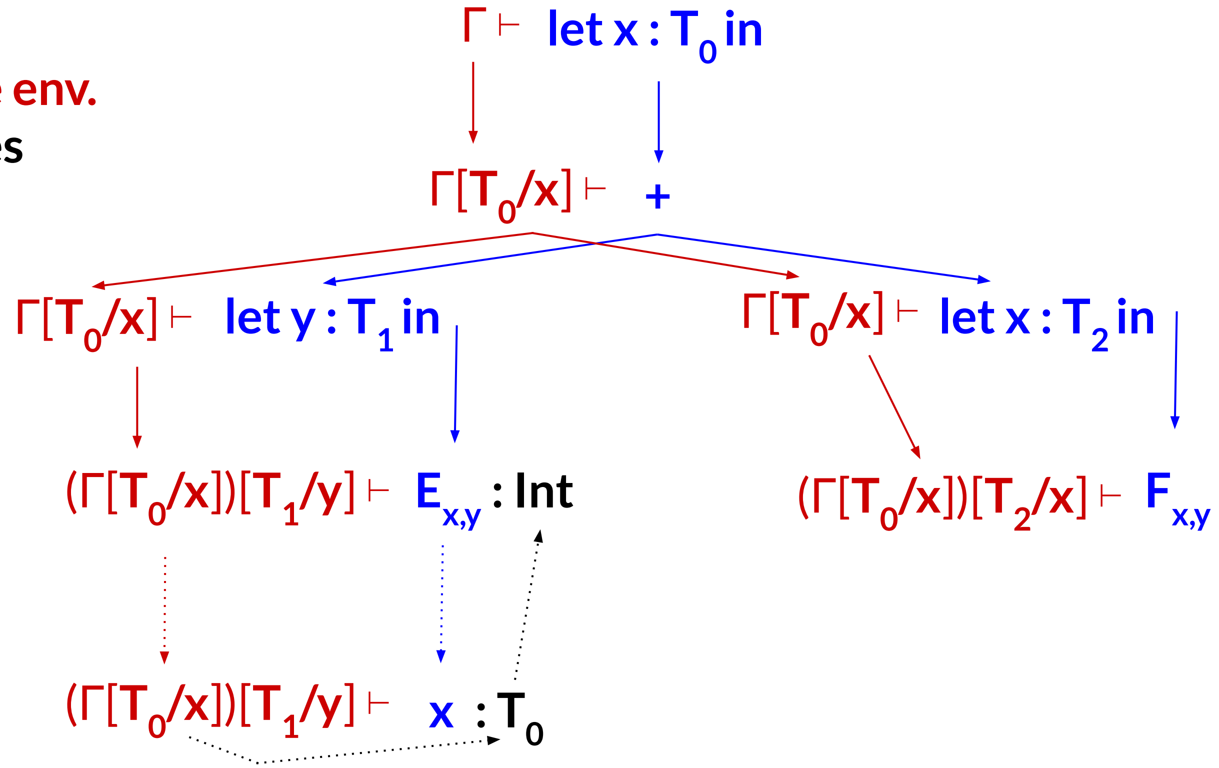
$(\Gamma[T_0/x])[T_1/y] \vdash E_{x,y}$

$(\Gamma[T_0/x])[T_2/x] \vdash F_{x,y}$

$(\Gamma[T_0/x])[T_1/y] \vdash x : T_0$

Example of Typing Let

AST
Type env.
Types



Example of Typing Let

AST
Type env.
Types

$$\Gamma \vdash \text{let } x : T_0 \text{ in}$$

$$\Gamma[T_0/x] \vdash +$$

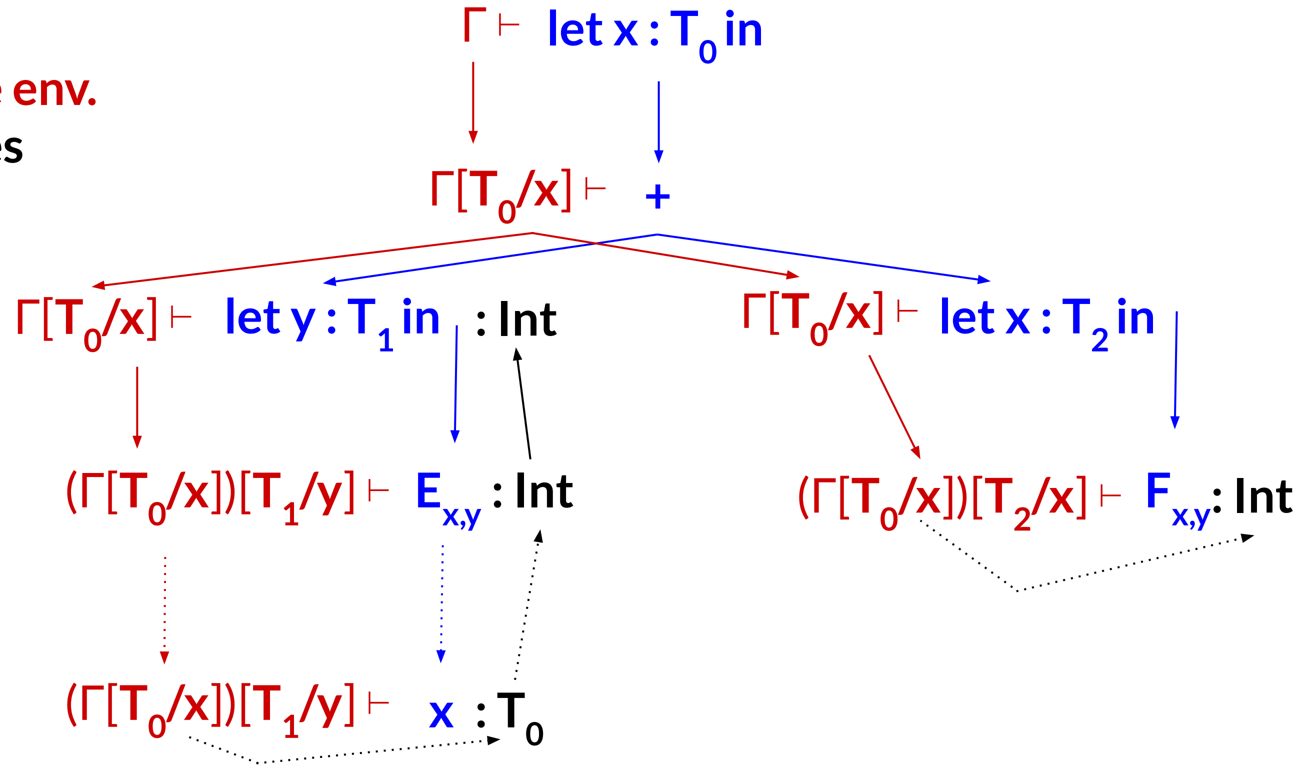
$$\Gamma[T_0/x] \vdash \text{let } y : T_1 \text{ in} \quad : \text{Int} \qquad \Gamma[T_0/x] \vdash \text{let } x : T_2 \text{ in}$$

$$(\Gamma[T_0/x])[T_1/y] \vdash E_{x,y} : \text{Int} \qquad (\Gamma[T_0/x])[T_2/x] \vdash F_{x,y}$$

$$(\Gamma[T_0/x])[T_1/y] \vdash x : T_0$$

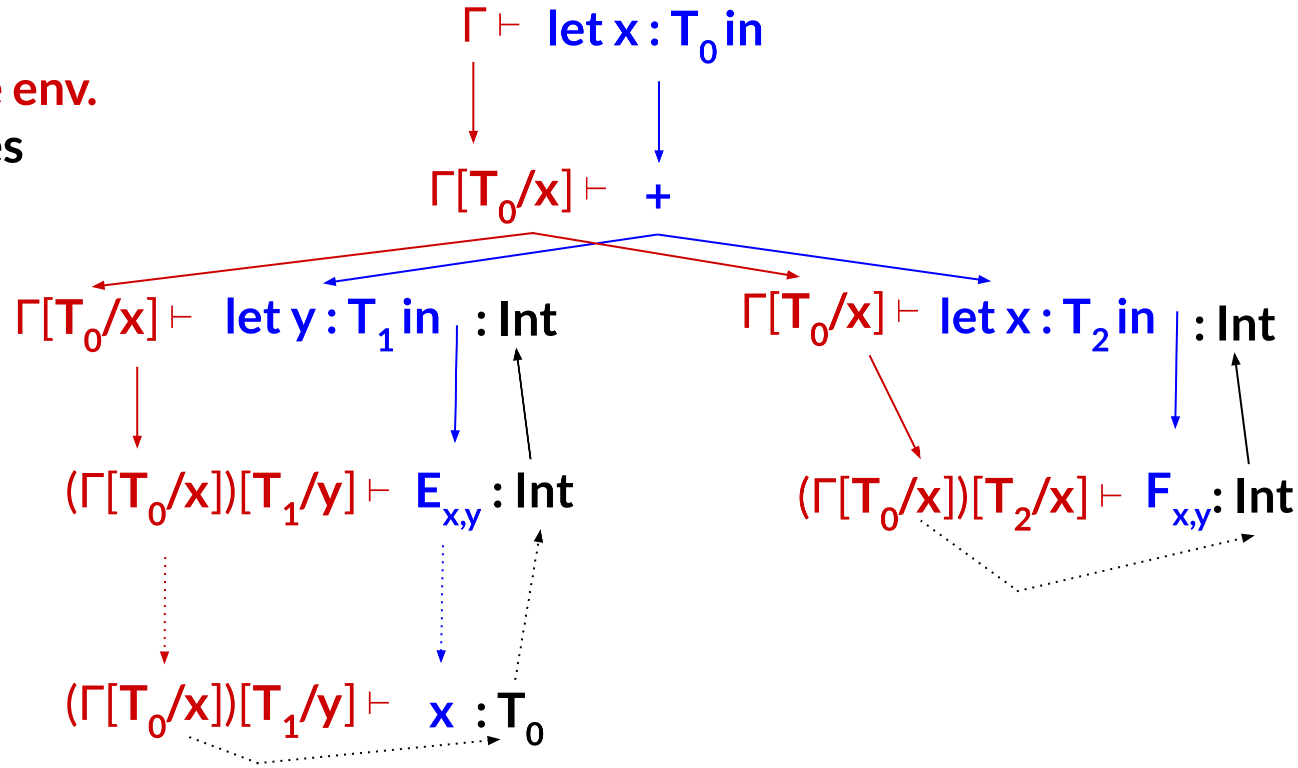
Example of Typing Let

AST
Type env.
Types

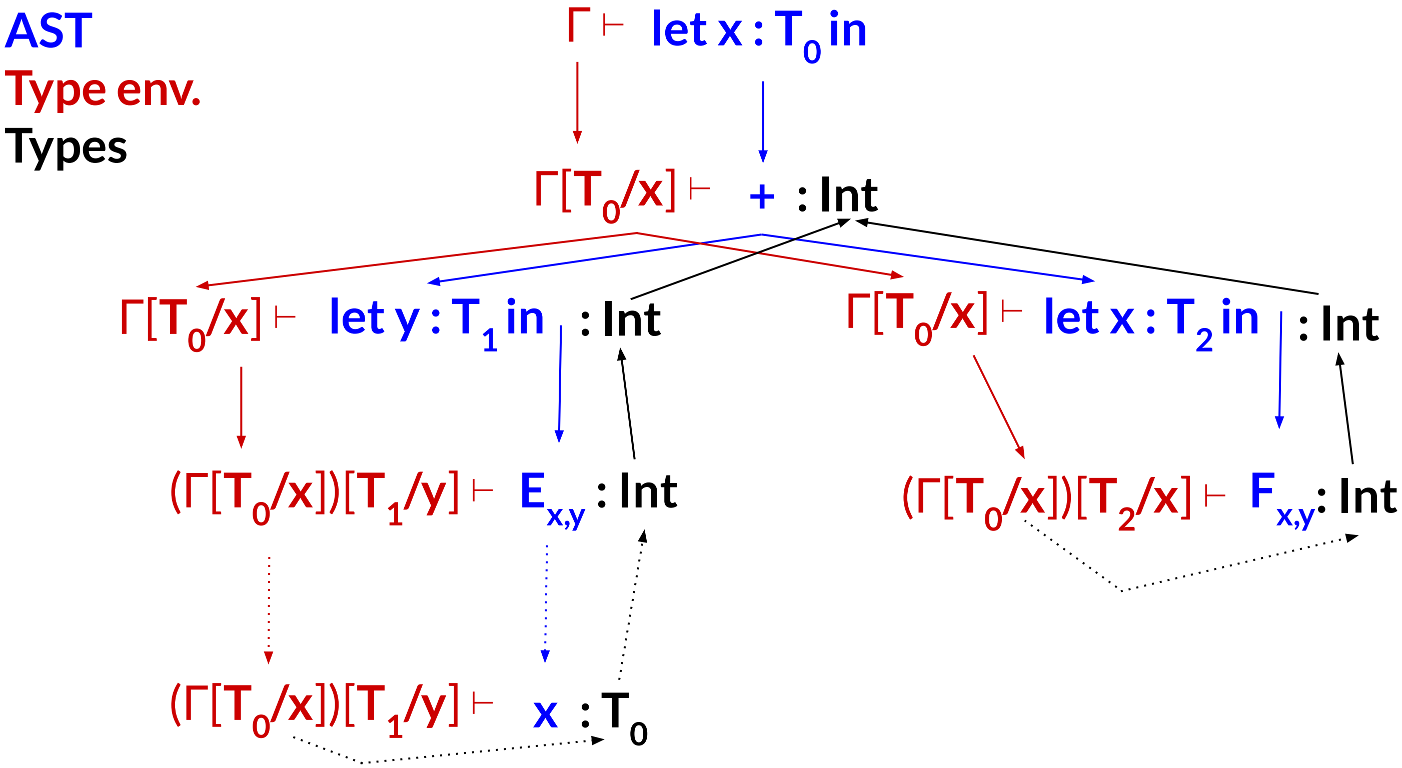


Example of Typing Let

AST
Type env.
Types

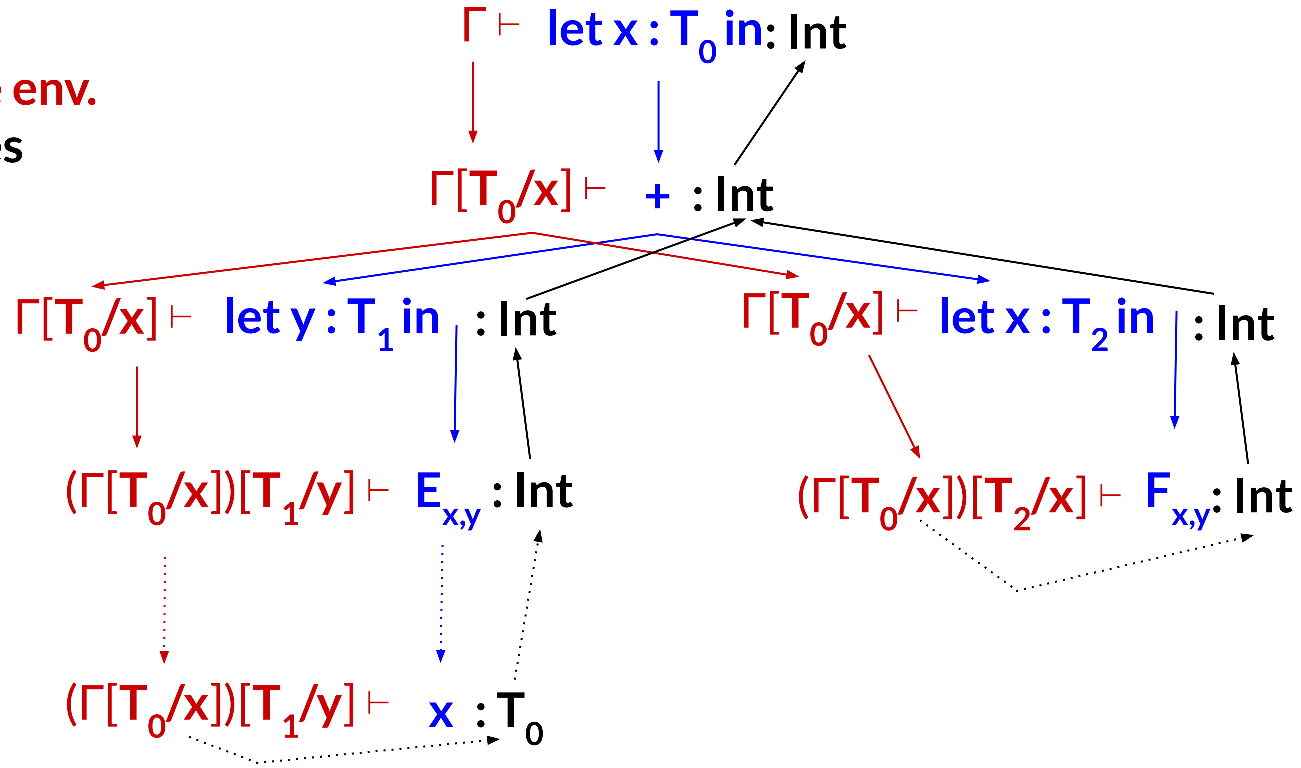


Example of Typing Let



Example of Typing Let

AST
Type env.
Types



More Let Practice

- Consider **1+ let x : Int in x + 2**

More Let Practice

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

More Let Practice

- Consider **1+ let x : Int in x + 2**
- What would the typing derivation be? Get out a piece of paper. I'll get you started...

$$\frac{???}{\Gamma \vdash 1 + \textbf{let x : Int in x + 2} : ???}$$

Type Environment Notes

- The type environment gives types to the **free variables** (only) in the current scope

Type Environment Notes

- The type environment gives types to the **free variables** (only) in the current scope
- The type environment is **passed down** the AST from the root towards the leaves

Type Environment Notes

- The type environment gives types to the **free variables** (only) 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**

Trivia Break: History

This book documented the environmental harm caused by the indiscriminate use of DDT, a pesticide used by soldiers during World War II. It was met with fierce opposition by chemical companies, but it swayed public opinion and led to a reversal in US pesticide policy, a nationwide ban on DDT for agricultural uses, and an environmental movement that led to the creation of the US Environmental Protection Agency. The book's author is Rachel Carson.

Let with Initialization

- Now consider let **with initialization**:

$$\frac{\Gamma \vdash e_0 : T_0 \qquad\qquad \Gamma[T_0/x] \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \texttt{ <- } e_0 \textbf{ in } e_1 : T_1} \textbf{[Let-Init]}$$

Let with Initialization

- Now consider let **with initialization**:

$$\frac{\Gamma \vdash e_0 : T_0 \qquad \Gamma[T_0/x] \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \textbf{ <- } e_0 \textbf{ in } e_1 : T_1} \; \textbf{[Let-Init]}$$

- This rule is **weak**. Why?

Let with Initialization

$$\frac{\Gamma \vdash e_0 : T_0 \qquad \Gamma[T_0/x] \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \textbf{ <- } e_0 \textbf{ in } e_1 : T_1}$$

Let with Initialization

$$\frac{\Gamma \vdash e_0 : T_0 \qquad \Gamma[T_0/x] \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \textbf{ <- } e_0 \textbf{ in } e_1 : T_1}$$

- To see why the rule is weak, consider this example:

```
class C inherits P { ... }
...
let x : P <- new C in ...
```

Let with Initialization

$$\frac{\Gamma \vdash e_0 : T_0 \qquad \Gamma[T_0/x] \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \textbf{ <- } e_0 \textbf{ in } e_1 : T_1}$$

- To see why the rule is weak, consider this example:

```
class C inherits P { ... }
...
let x : P <- new C in ...
```

- The proposed let rule **does not allow** this code (because "new C" (e_0) does not exactly have the type "P" (T_0))

Let with Initialization

$$\frac{\Gamma \vdash e_0 : T_0 \qquad \Gamma[T_0/x] \vdash e_1 : T_1}{\Gamma \vdash \text{let } x : T_0 \text{ <- } e_0 \text{ in } e_1 : T_1}$$

- To see why the rule is weak, consider this example:

  class C inherits P { ... }
  ...
  let x : P <- new C in ...

- The proposed let rule **does not allow** this code (because "new C" (e_0) does not exactly have the type "P" (T_0))
  - We say that a rule is ***too weak*** or ***incomplete*** when it prevents us from typechecking "good" programs (but Rice's Theorem...)

Subtyping

Subtyping

- Define a relation **X ≤ Y** on classes that denotes:

Subtyping

Reminder from math class: a ***relation*** is some specific subset of the Cartesian product of some finite list of sets

- Define a <u>relation</u> **X ≤ Y** on classes that denotes:

Subtyping

- Define a relation **$X \leq Y$** on classes that denotes:

Subtyping

- Define a relation **X ≤ Y** on classes that denotes:
  - an object of type **X** can be used when one of type **Y** is expected (***Liskov substitutability***), or equivalently

Subtyping

- Define a relation **X ≤ Y** on classes that denotes:
  - an object of type **X** can be used when one of type **Y** is expected (***Liskov substitutability***), or equivalently
  - **X** ***conforms*** with **Y**

Subtyping

- Define a relation **X ≤ Y** on classes that denotes:
    - an object of type **X** can be used when one of type **Y** is expected (***Liskov substitutability***), or equivalently
    - **X** *conforms* with **Y**
    - In Cool, this means that **X** is a ***subclass*** of **Y**

Subtyping

- Define a relation **X ≤ Y** on classes that denotes:
  - an object of type **X** can be used when one of type **Y** is expected (***Liskov substitutability***), or equivalently
  - **X** *conforms* with **Y**
  - In Cool, this means that **X** is a ***subclass*** of **Y**
- Definition of **≤** on classes:
  - **X ≤ X**
  - **X ≤ Y** if **X** inherits from **Y**
  - **X ≤ Z** if **X ≤ Y** and **Y ≤ Z**

Let with Initialization (better)

Let with Initialization (better)

$$\frac{\Gamma \vdash e_0 : T \qquad T \leq T_0 \qquad \Gamma[T_0/x] \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \textbf{ <- } e_0 \textbf{ in } e_1 : T_1} \textbf{ [Let-Init]}$$

Let with Initialization (better)

$$\frac{\Gamma \vdash e_0 : T \qquad T \leq T_0 \qquad \Gamma[T_0/x] \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \textbf{ <- } e_0 \textbf{ in } e_1 : T_1} \textbf{ [Let-Init]}$$

- Both rules are ***sound***
  - I.e., neither rule allows any “bad” programs to typecheck

Let with Initialization (better)

$$\frac{\Gamma \vdash e_0 : T \qquad T \leq T_0 \qquad \Gamma[T_0/x] \vdash e_1 : T_1}{\Gamma \vdash \textbf{let}\ x : T_0 \ \texttt{<-}\ e_0\ \textbf{in}\ e_1 : T_1}\ \textbf{[Let-Init]}$$

- Both rules are ***sound***
  - I.e., neither rule allows any “bad” programs to typecheck
- But more programs typecheck with this new rule
  - It is more ***complete***

Expressiveness of Type Systems

- There is a tension between:
  - flexible rules that **do not constrain** programmers
  - restrictive rules that **ensure safety** of execution

Expressiveness of Type Systems

- There is a tension between:
  - flexible rules that **do not constrain** programmers
  - restrictive rules that **ensure safety** of execution
- This tension is necessitated by ***Rice's Theorem***: all non-trivial semantic properties of a program are undecidable.

Expressiveness of Type Systems

- There is a tension between:
  - flexible rules that **do not constrain** programmers
  - restrictive rules that **ensure safety** of execution
- This tension is necessitated by ***Rice's Theorem***: all non-trivial semantic properties of a program are undecidable.
  - To make the typechecker decidable, we **must compromise** on either soundness or completeness

Expressiveness of Type Systems

- There is a tension between:
    - flexible rules that **do not constrain** programmers
    - restrictive rules that **ensure safety** of execution
- This tension is necessitated by ***Rice's Theorem***: all non-trivial semantic properties of a program are undecidable.
    - To make the typechecker decidable, we **must compromise** on either soundness or completeness
        - i.e., we must either reject some "good" programs or accept some "bad" programs

Expressiveness of Type Systems

- There is a tension between:
  - flexible rules that **do not constrain** programmers
  - restrictive rules that **ensure safety** of execution
- This tension is necessitated by ***Rice's Theorem***: all non-trivial semantic properties of a program are undecidable.
  - To make the typechecker decidable, we **must compromise** on either soundness or completeness
    - i.e., we must either reject some "good" programs or accept some "bad" programs
- **Usual choice** (for static type systems): reject some "good" programs

Static vs Dynamic Types (again)

- **Usual choice** (for static type systems): reject some "good" programs

Static vs Dynamic Types (again)

- **Usual choice** (for static type systems): reject some “good” programs
  - Advantage: no “bad” programs will get executed!

Static vs Dynamic Types (again)

- **Usual choice** (for static type systems): reject some “good” programs
  - Advantage: no “bad” programs will get executed!
- This choice is a **big downside** of static type systems - they impose a **cost** on the user by disallowing some correct programs

Static vs Dynamic Types (again)

- **Usual choice** (for static type systems): reject some “good” programs
  - Advantage: no “bad” programs will get executed!
- This choice is a **big downside** of static type systems - they impose a **cost** on the user by disallowing some correct programs
  - some argue for **dynamic typechecking** instead
    - “compute is cheap, human attention is expensive”

Static vs Dynamic Types (again)

- **Usual choice** (for static type systems): reject some “good” programs
  - Advantage: no “bad” programs will get executed!
- This choice is a **big downside** of static type systems - they impose a **cost** on the user by disallowing some correct programs
  - some argue for **dynamic typechecking** instead
    - “compute is cheap, human attention is expensive”
  - others argue for **more expressive** static typechecking instead
    - however, more expressive type systems are **more complex**

Soundness

- The notions of **static types** and **dynamic types** are necessary for stating the ***soundness theorems*** of advanced type systems like Cool's

Soundness

- The notions of **static types** and **dynamic types** are necessary for stating the ***soundness theorems*** of advanced type systems like Cool's
  - a type system's soundness theorem proves that it rejects all "bad" programs

Soundness

- The notions of **static types** and **dynamic types** are necessary for stating the ***soundness theorems*** of advanced type systems like Cool's
  - a type system's soundness theorem proves that it rejects all "bad" programs
- Define the *dynamic type* of an object as the class C that is used in the "new C" expression that creates the object in some execution

Soundness

- The notions of **static types** and **dynamic types** are necessary for stating the ***soundness theorems*** of advanced type systems like Cool's
  - a type system's soundness theorem proves that it rejects all "bad" programs
- Define the ***dynamic type*** of an object as the class C that is used in the "new C" expression that creates the object in some execution
  - run-time notion, present even in languages without static types

Soundness

- The notions of **static types** and **dynamic types** are necessary for stating the ***soundness theorems*** of advanced type systems like Cool's
  - a type system's soundness theorem proves that it rejects all "bad" programs
- Define the ***dynamic type*** of an object as the class C that is used in the "new C" expression that creates the object in some execution
  - run-time notion, present even in languages without static types
- Define the ***static type*** of an expression as the *least upper bound* of the dynamic types that the expression can take on, in some execution
  - cf. static vs dynamic semantics

Soundness

- The notions of **static types** and **dynamic types** are necessary for stating the ***soundness theorems*** of advanced type systems like Cool's
    - a type system's soundness theorem proves that it rejects all "bad" programs
- Define the ***dynamic type*** of an objec[…] "new C" expression that creates the […]
    - run-time notion, present even in languages without static types
- Define the ***static type*** of an expression as the *least upper bound* of the dynamic types that the expression can take on, in some execution
    - cf. static vs dynamic semantics

> Think of ***least upper bound*** as "nearest common ancestor in the type hierarchy", for now

Stating a Soundness Theorem

- In early type systems, the set of static types **corresponded exactly** with the dynamic types

Stating a Soundness Theorem

- In early type systems, the set of static types **corresponded exactly** with the dynamic types
  - in such a type system, the soundness theorem is easy to state:

    for all expressions E, dynamic_type(E) = static_type(E)

Stating a Soundness Theorem

- In early type systems, the set of static types **corresponded exactly** with the dynamic types
    - in such a type system, the soundness theorem is easy to state:

      for all expressions E, dynamic_type(E) = static_type(E)
- This gets more complicated for “advanced” type systems (Java, Cool)
    - Why?

Stating a Soundness Theorem

- In early type systems, the set of static types **corresponded exactly** with the dynamic types
  - in such a type system, the soundness theorem is easy to state:

    for all expressions E, dynamic_type(E) = static_type(E)
- This gets more complicated for "advanced" type systems (Java, Cool)
  - Why? Must also consider **subtyping** / Liskov substitutability

Stating a Soundness Theorem

- In early type systems, the set of static types **corresponded exactly** with the dynamic types
  - in such a type system, the soundness theorem is easy to state:

    for all expressions E, dynamic_type(E) = static_type(E)
- This gets more complicated for “advanced” type systems (Java, Cool)
  - Why? Must also consider **subtyping** / Liskov substitutability
  - so, Cool’s soundness theorem is:

    for all expressions E, dynamic_type(E) ≤ static_type(E)

Stating a Soundness Theorem

Why is this soundness theorem okay?

for all expressions E, dynamic_type(E) ≤ static_type(E)

Stating a Soundness Theorem

Why is this soundness theorem okay?

- For all E, the compiler allows only operations that **static_type(E)** permits

for all expressions E, dynamic_type(E) ≤ static_type(E)

Stating a Soundness Theorem

Why is this soundness theorem okay?

- For all E, the compiler allows only operations that **static_type(E)** permits
- Liskov substitutability guarantees that any operation available on a supertype is also available on its subtypes

for all expressions E, dynamic_type(E) ≤ static_type(E)

Stating a Soundness Theorem

- In

Why is this soundness theorem okay?

- For all E, the compiler allows only operations that **static_type(E)** permits
- Liskov substitutability guarantees that any operation available on a supertype is also available on its subtypes
  - subclasses can **only add** attributes or methods

for all expressions E, dynamic_type(E) ≤ static_type(E)

Stating a Soundness Theorem

Why is this soundness theorem okay?

- For all E, the compiler allows only operations that **static_type(E)** permits
- Liskov substitutability guarantees that any operation available on a supertype is also available on its subtypes
  - subclasses can **only add** attributes or methods
  - methods can be redefined, but **only with the same types**

for all expressions E, dynamic_type(E) ≤ static_type(E)

Subtyping Example

- Consider the following Cool classes:

  class A { a() : Int { 0 }; }
  class B inherits A { b() : Int { 1 }; }

Subtyping Example

- Consider the following Cool classes:

  ```
  class A { a() : Int { 0 }; }
  class B inherits A { b() : Int { 1 }; }
  ```

- An instance of **B** has methods “a()” and “b()”

Subtyping Example

- Consider the following Cool classes:

  class A { a() : Int { 0 }; }
  class B inherits A { b() : Int { 1 }; }

- An instance of **B** has methods "a()" and "b()"
- And instance of **A** only has method "a()"

Subtyping Example

- Consider the following Cool classes:

  class A { a() : Int { 0 }; }
  class B inherits A { b() : Int { 1 }; }

- An instance of **B** has methods "a()" and "b()"
- And instance of **A** only has method "a()"
  - A type error will occur if we try to invoke method "b()" on an object with dynamic type A

Subtyping Example

- Consider the following Cool classes:

  class A { a() : Int { 0 }; }
  class B inherits A { b() : Int { 1 }; }

- An instance of **B** has methods "a()" and "b()"
- And instance of **A** only has method "a()"
  - A type error will occur if we try to invoke method "b()" on an object with dynamic type A
  - But the static type system will **forbid** such an invocation!

Example of Wrong Let Rule (1)

Examples of Wrong Let Rule (1)

- Now consider a hypothetical **wrong** let rule:

$$\frac{\Gamma \vdash e_0 : T \qquad T \leq T_0 \qquad \Gamma \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \textbf{ <- } e_0 \textbf{ in } e_1 : T_1} \textbf{ [Let-Init]}$$

- How is it different from the correct rule?

Examples of Wrong Let Rule (1)

- Now consider a hypothetical **wrong** let rule:

$$\frac{\Gamma \vdash e_0 : T \qquad T \leq T_0 \qquad \Gamma \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \textbf{ <- } e_0 \textbf{ in } e_1 : T_1} \textbf{ [Let-Init]}$$

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

  let x : Int <- 0 in x + 1

- Why not?

Examples of Wrong Let Rule (1)

- Now consider a hypothetical **wrong** let rule:

$$\frac{\Gamma \vdash e_0 : T \qquad T \leq T_0 \qquad \Gamma \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \textbf{ <- } e_0 \textbf{ in } e_1 : T_1} \textbf{[Let-Init]}$$

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

  let x : Int <- 0 in x + 1
- Why not? **Typing environment** hasn't been updated!

Examples of Wrong Let Rule (2)

- Now consider another hypothetical **wrong** let rule:

$$\frac{\Gamma \vdash e_0 : T \qquad T_0 \leq T \qquad \Gamma[T_0/x] \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \textbf{ <- } e_0 \textbf{ in } e_1 : T_1} \; \textbf{[Let-Init]}$$

- How is this one different from the correct rule?

Examples of Wrong Let Rule (2)

- Now consider another hypothetical **wrong** let rule:

$$\frac{\Gamma \vdash e_0 : T \qquad T_0 \leq T \qquad \Gamma[T_0/x] \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \textbf{ <- } e_0 \textbf{ in } e_1 : T_1} \textbf{ [Let-Init]}$$

- How is this one 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?

Examples of Wrong Let Rule (3)

- Now consider another hypothetical **wrong** let rule:

$$\frac{\Gamma \vdash e_0 : T \qquad T \leq T_0 \qquad \Gamma[T/x] \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \textbf{ <- } e_0 \textbf{ in } e_1 : T_1} \quad \textbf{[Let-Init]}$$

- How is this one different from the correct rule?

Examples of Wrong Let Rule (3)

- Now consider another hypothetical **wrong** let rule:

$$\frac{\Gamma \vdash e_0 : T \qquad T \leq T_0 \qquad \Gamma[T/x] \vdash e_1 : T_1}{\Gamma \vdash \textbf{let } x : T_0 \textbf{ <- } e_0 \textbf{ in } e_1 : T_1} \textbf{ [Let-Init]}$$

- How is this one different from the correct rule?
- This "good" program is not well-typed:

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

- Why isn't this program well-typed?

Type Rule Notation

- The type rules use **very concise** notation

Type Rule Notation

- The type rules use **very concise** notation
  - and they are very carefully constructed

Type Rule Notation

- The type rules use **very concise** notation
  - and they are very carefully constructed
- Virtually **any change** in a rule either:

Type Rule Notation

- The type rules use **very concise** notation
  - and they are very carefully constructed
- Virtually **any change** in a rule either:
  - Makes the type system **unsound**
    - i.e., bad programs are well-typed

Type Rule Notation

- The type rules use **very concise** notation
  - and they are very carefully constructed
- Virtually **any change** in a rule either:
  - Makes the type system **unsound**
    - i.e., bad programs are well-typed
  - Or, makes the type system less usable (more **incomplete**)
    - i.e., good programs are rejected

Type Rule Notation

- The type rules use **very concise** notation
    - and they are very carefully constructed
- Virtually **any change** in a rule either:
    - Makes the type system **unsound**
        - i.e., bad programs are well-typed
    - Or, makes the type system less usable (more incomplete)
        - i.e., good programs are rejected
- But no matter how well we choose the type rules, ***some good programs will be rejected anyway***

Type Rule Notation

- The type rules use **very concise** notation
  - and they are very carefully constructed
- Virtually **any change** in a rule either:
  - Makes the type system **unsound**
    - i.e., bad programs are well-typed
  - Or, makes the type system less usable (more **incomplete**)
    - i.e., good programs are rejected
- But no matter how well we choose the type rules, ***some good programs will be rejected anyway***
  - Rice's Theorem strikes again: typechecking is **undecidable**

Type Rule Notation

- The type rules use **very concise** n
  - and they are very carefully co
- Virtually **any change** in a rule eit
  - Makes the type system **unsou**
    - i.e., bad programs are well-typed
  - Or, makes the type system less usable (more **incomplete**)
    - i.e., good programs are rejected
- But no matter how well we choose the type rules, ***some good programs will be rejected anyway***
  - Rice's Theorem strikes again: typechecking is **undecidable**

Next time we'll cover some **even-more-complex rules** than let:

- Typechecking method dispatch
- Typechecking with SELF_TYPE in Cool

Course Announcements

- Don’t put off starting PA2!
  - This part of the semester may feel like a lull, but that feeling isn’t accurate!