Derivations

Concepts of Programming Languages Lecture 5

Practice Problem

What is the typing rule for let expression?

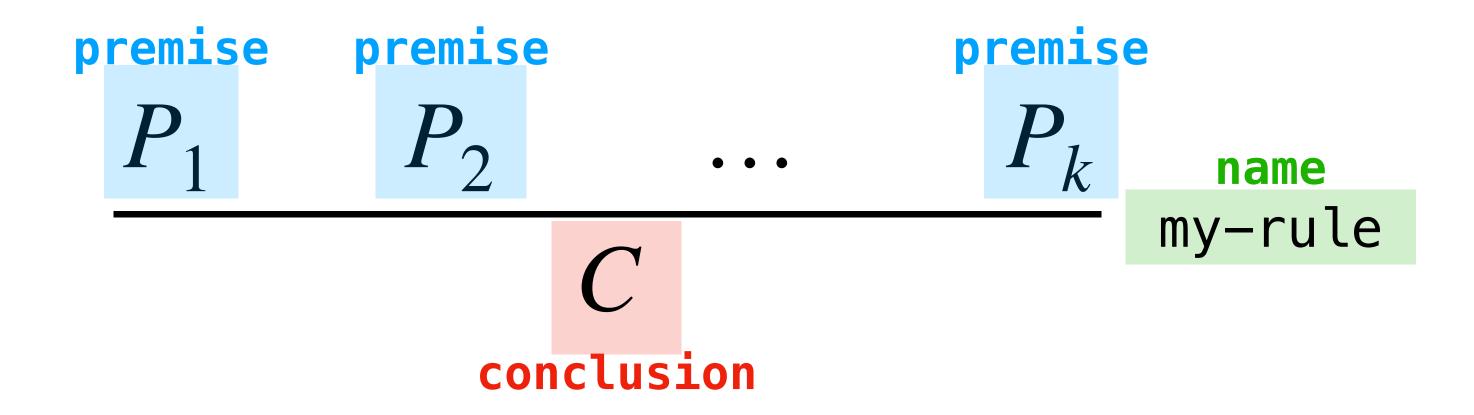
What are the semantics rules for if expression?

Outline

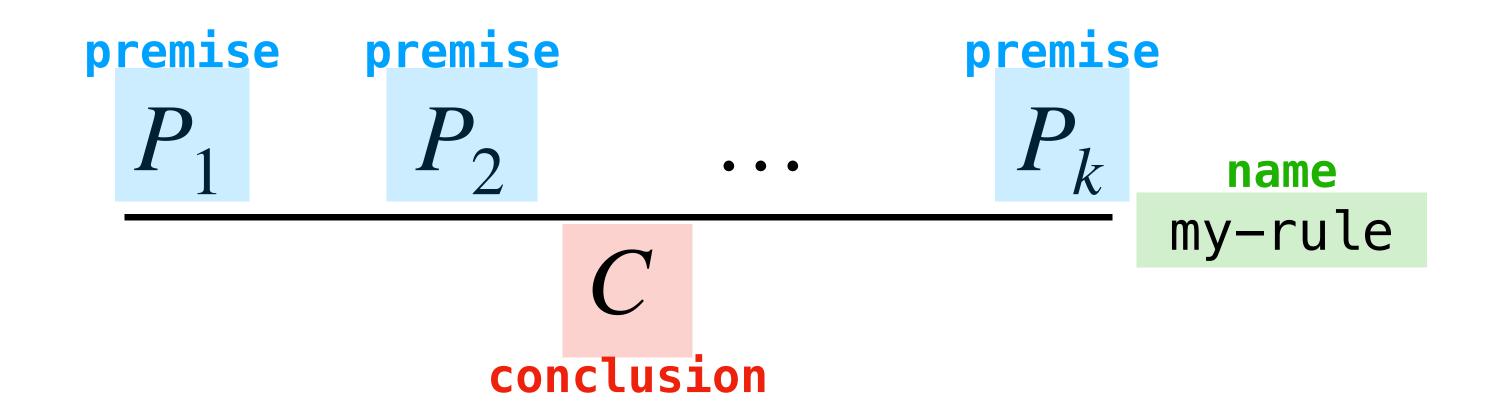
- » Discuss derivations in general
- » See how to read and write derivations
- » Go through a couple examples

Recap

Recall: Inference Rules

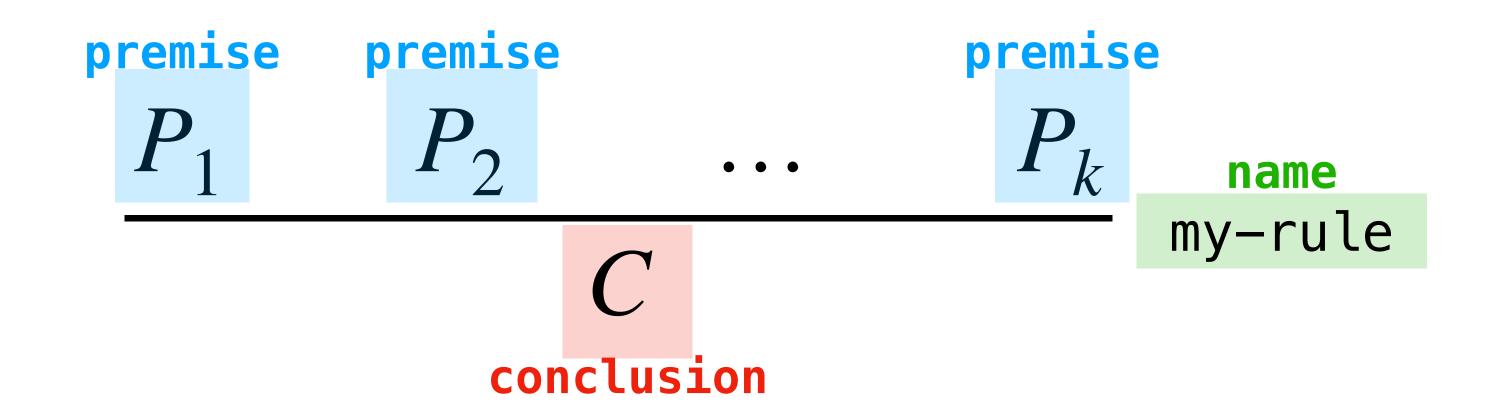


Recall: Inference Rules



Then general form of an inference rule has a collection of **premises** and a **conclusion**. Read as

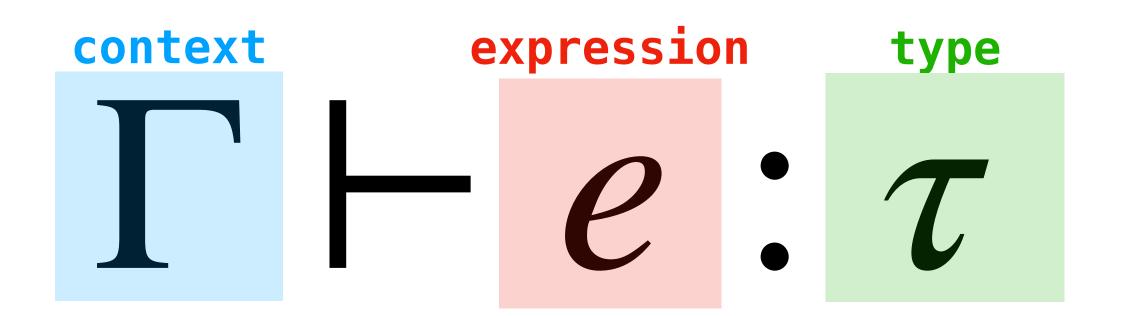
Recall: Inference Rules



Then general form of an inference rule has a collection of **premises** and a **conclusion**. Read as

If P_1 holds and P_2 holds and ... P_k holds, then C holds (by my-rule)

Recall: Typing Judgments



A <u>typing judgment</u> is a compact way of representing the statement:

e is of type τ in/under the context Γ

```
\Gamma = \{ x : int, y : string, z : int -> string \}
```

```
\Gamma = \{ x : int, y : string, z : int -> string \}
```

A context is a set of variable declarations

 $\Gamma = \{ x : int, y : string, z : int -> string \}$

A context is a set of variable declarations

A variable declaration $(x : \tau)$ says: "I declare that the variable x is of type τ "

 $\Gamma = \{ x : int, y : string, z : int -> string \}$

A context is a set of variable declarations

A variable declaration $(x : \tau)$ says: "I declare that the variable x is of type τ "

A context keeps track of all the types of variables in the "environment"

Derivations

```
 \frac{ }{ \{\} \vdash 2 : int \}} (intLit) \frac{ }{ \{y : int\} \vdash y : int \}} (var) \frac{ }{ \{y : int\} \vdash y : int \}} (intAdd) }{ \{y : int\} \vdash y + y : int } (let) 
 \{\} \vdash let \ y = 2 \ in \ y + y : int \}
```

Derivations allow us to *prove* that a typing judgment holds with respect to a collection of inference rules

```
 \frac{ \frac{}{\{y : int\} \vdash y : int} (var) }{\{y : int\} \vdash y : int} \frac{\{y : int\} \vdash y : int}{\{y : int\} \vdash y : int} (intAdd) }{\{y : int\} \vdash y + y : int} (let)
```

Derivations allow us to *prove* that a typing judgment holds with respect to a collection of inference rules

Formally, a derivation is a tree in which:

```
 \frac{ \frac{}{\{y: int\} \vdash y: int} (var) }{\{y: int\} \vdash y: int} \frac{\{y: int\} \vdash y: int}{\{y: int\} \vdash y: int} (intAdd) }{\{y: int\} \vdash y: int} 
 \frac{\{y: int\} \vdash y: int}{\{y: int\} \vdash y: int} (let) }{\{y: int\} \vdash y: int}
```

Derivations allow us to *prove* that a typing judgment holds with respect to a collection of inference rules

Formally, a derivation is a tree in which:

» each node is labeled with a typing judgment

Derivations allow us to *prove* that a typing judgment holds with respect to a collection of inference rules

Formally, a derivation is a tree in which:

- » each node is labeled with a typing judgment
- » and typing judgment follows from the typing judgments at it's children by an inference rule

Applying Rules
$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} \text{ (let)} \quad \frac{\Gamma \vdash e_1 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \text{ (intAdd)}$$

```
-\frac{(\text{intLit})}{\{y:\text{int}\}\vdash y:\text{int}}\frac{\{y:\text{int}\}\vdash y:\text{int}}{\{y:\text{int}\}\vdash y:\text{int}}\frac{(\text{var})}{(\text{intAdd})}}{\{y:\text{int}\}\vdash y:\text{int}}\frac{\{y:\text{int}\}\vdash y:\text{int}}{\{y:\text{int}\}}
\{\} \vdash 2 : \mathtt{int}
```

Applying Rules
$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} \text{ (let)} \quad \frac{\Gamma \vdash e_1 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \text{ (intAdd)}$$

```
\frac{\{y: int\} \vdash y: int}{\{y: int\} \vdash y: int} \frac{\{y: int\} \vdash y: int}{\{y: int\} \vdash y: int} (var)}{\{y: int\} \vdash y: int} (intAdd)
\{\} \vdash let \ y = 2 \ in \ y + y: int}
```

So far, we've used rules as ways of describing the behavior of a PL

Applying Rules
$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau} \text{ (let)} \quad \frac{\Gamma \vdash e_1 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \text{ (intAdd)}$$

```
\frac{}{\{\}\vdash 2: \mathtt{int}}(\mathtt{intLit}) \quad \frac{}{\{y: \mathtt{int}\}\vdash y: \mathtt{int}}(\mathtt{var}) \quad \frac{}{\{y: \mathtt{int}\}\vdash y: \mathtt{int}}(\mathtt{var})} \\ = \frac{}{\{\}\vdash \mathtt{let} \ y = 2 \ \mathtt{in} \ y + y: \mathtt{int}}(\mathtt{let})}
```

So far, we've used rules as ways of describing the behavior of a PL

When we build typing derivations, we instantiate the meta-variables in the rule at particular expressions, contexts, etc.

```
 \frac{ }{ \{\} \vdash 2 : int \}} (intLit) \frac{ }{ \{y : int\} \vdash y : int \}} (var) \frac{ }{ \{y : int\} \vdash y : int \}} (intAdd) }{ \{y : int\} \vdash y + y : int } (let) 
 \{\} \vdash let \ y = 2 \ in \ y + y : int \}
```

1. Start from the bottom!

```
 \frac{ \frac{ }{\{\} \vdash 2 : \mathtt{int} } (\mathtt{intLit}) \quad \frac{ \frac{ }{\{y : \mathtt{int}\} \vdash y : \mathtt{int} } (\mathtt{var}) \quad \frac{ }{\{y : \mathtt{int}\} \vdash y : \mathtt{int} } (\mathtt{intAdd}) }{ \frac{ }{\{y : \mathtt{int}\} \vdash y + y : \mathtt{int} } (\mathtt{let}) }
```

- 1. Start from the bottom!
- 2. Apply the rule based on the expression

- 1. Start from the bottom!
- 2. Apply the rule based on the expression

Repeat for the premises

Axioms (When are we done?)

```
 \frac{ }{ \{\} \vdash 2 : int \}} (intLit) \frac{ }{ \{y : int\} \vdash y : int \}} (var) \frac{ }{ \{y : int\} \vdash y : int \}} (intAdd) }{ \{y : int\} \vdash y + y : int } (let) 
 \{\} \vdash let \ y = 2 \ in \ y + y : int \}
```

Axioms (When are we done?)

We know that we can stop building a derivation once we need to derive a premise with an **axiom**, i.e., a rule with no premises

Axioms (When are we done?)

```
\frac{\frac{}{\{\}\vdash 2: \mathtt{int}}(\mathtt{intLit})}{\frac{\{y: \mathtt{int}\}\vdash y: \mathtt{int}}{\{y: \mathtt{int}\}\vdash y: \mathtt{int}}}(\mathtt{var})} \frac{}{\{y: \mathtt{int}\}\vdash y: \mathtt{int}}}(\mathtt{intAdd})}{\{y: \mathtt{int}\}\vdash y: \mathtt{int}}}{\{\}\vdash \mathtt{let}\ y = 2\ \mathtt{in}\ y+y: \mathtt{int}}}(\mathtt{let})
```

We know that we can stop building a derivation once we need to derive a premise with an **axiom**, i.e., a rule with no premises

In our case, this will almost always be "literal" or "variable" rules

```
 \frac{ }{ \{\} \vdash 2 : int \}} (intLit) \frac{ }{ \{y : int\} \vdash y : int \}} (var) \frac{ }{ \{y : int\} \vdash y : int \}} (intAdd) }{ \{y : int\} \vdash y + y : int } (let) 
 \{\} \vdash let \ y = 2 \ in \ y + y : int \}
```

1. Start from the bottom!

- 1. Start from the bottom!
- 2. Apply the rule based on the expression

```
 \frac{ \frac{}{\{\} \vdash 2 : \text{int}} (\text{intLit}) \quad \frac{\overline{\{y : \text{int}\} \vdash y : \text{int}} \quad \overline{\{y : \text{int}\} \vdash y : \text{int}}}{\{y : \text{int}\} \vdash y + y : \text{int}} (\text{intAdd}) } }{\{\} \vdash \text{let } y = 2 \text{ in } y + y : \text{int}} (\text{let}) }
```

- 1. Start from the bottom!
- 2. Apply the rule based on the expression

Repeat for premises until there are none left

```
 \frac{ }{ \{\} \vdash 2 : int \}} (intLit) \frac{ }{ \{y : int\} \vdash y : int \}} (var) \frac{ }{ \{y : int\} \vdash y : int \}} (intAdd) }{ \{y : int\} \vdash y + y : int } (let) 
 \{\} \vdash let \ y = 2 \ in \ y + y : int \}
```

```
 \frac{ \frac{ }{\{\} \vdash 2 : int} (intLit) }{ \frac{\{y : int\} \vdash y : int}{\{y : int\} \vdash y : int} (var) }{ \frac{\{y : int\} \vdash y : int}{\{y : int\}} (let) }
```

Easy because of a very cool feature of OCaml:

```
\frac{ \frac{}{\{\} \vdash 2 : int\}} (intLit)}{\{\} \vdash 2 : int\}} \frac{\frac{}{\{y : int\} \vdash y : int}}{\{y : int\} \vdash y : int}} \frac{(var)}{\{y : int\} \vdash y : int}} (let)
\{\} \vdash let \ y = 2 \ in \ y + y : int}
```

Easy because of a very cool feature of OCaml:

Syntax Directed Type System

```
 \frac{ }{ \left\{ \right\} \vdash 2 : int} (intLit) \quad \frac{ \overline{\left\{ y : int \right\} \vdash y : int} \quad \overline{\left\{ y : int \right\} \vdash y : int} \quad (var) }{ \left\{ y : int \right\} \vdash y + y : int} (let) } 
 \left\{ \right\} \vdash 1et \quad y = 2 \quad in \quad y + y : int
```

Easy because of a very cool feature of OCaml:

Syntax Directed Type System

Only one typing rule per expression.

Integer Literals

1. $\frac{\text{n is an int lit}}{\Gamma \vdash \text{n : int}} \text{ (intLit)} \qquad \frac{\text{n is an int lit}}{\text{n } \psi n} \text{ (intLitEval)}$

- 1. If n is an integer literal, then it is of type int in any context
- 2. If **n** is an integer literal, then it evaluates to the number it represents

A Note about Side Conditions

If a premise is a side-condition this it is not included in the derivation

Side conditions need to hold in order to apply the rule, but they don't appear in the derivation itself

We will try to always write side conditions in green

Float Literals

- 1. $\frac{\text{n is a float lit}}{\Gamma \vdash \text{n : float}} \text{ (floatLit)} \qquad \frac{\text{n is a float lit}}{\text{n} \Downarrow n} \text{ (floatLitEval)}$
 - 1. If n is an float literal, then it is of type float in any context
 - 2. If n is an float literal, then it evaluates to the number it represents

Boolean Literals

- 1. $\frac{1}{\Gamma \vdash \text{true : bool}} \text{ (trueLit)}$

- 2. $\frac{1}{\Gamma \vdash \text{false : bool}} \text{ (falseLit)}$
- 1. true is of type bool in any context
- 2. false if of type bool in any context
- 3. true evaluates to the value T
- 4. false evaluates to the value ⊥

Variables

$$\frac{(v:\tau) \in \Gamma}{\Gamma \vdash v:\tau} \text{ (intLit)}$$

If v is declared to be of type τ in the context Γ , then v is of type τ in Γ

Variables cannot be evaluated (more on this when we talk about substitution and well-scopedness)

Okay, that was a lot, let's do some examples

Back to the Example

Back to the Example

A derivation is just a mathy way of writing a natural language prove that a typing derivation holds

Back to the Example

A derivation is just a mathy way of writing a natural language prove that a typing derivation holds

(In fact, most mathematical arguments can be represented formally as derivation trees, this is the called **proof theory**)

```
\frac{}{ \{\} \vdash 2 : int \}} (intLit) = \frac{}{\{y : int\} \vdash y : int \}} (var) = \frac{}{\{y : int\} \vdash y : int \}} (var) = \frac{}{\{y : int\} \vdash y + y : int \}} (let)
\{\} \vdash let \ y = 2 \ in \ y + y : int \}
```

```
\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let} \ x = e_1 \ \mathsf{in} \ e_2 : \tau_2} \ (\mathsf{let})
```

```
\frac{}{ \{\} \vdash 2 : int \}} (intLit) = \frac{}{\{y : int\} \vdash y : int} (var) - \frac{}{\{y : int\} \vdash y : int} (var) - \frac{}{\{y : int\} \vdash y : int} (intAdd) - \frac{}{\{\} \vdash let \ y = 2 \ in \ y + y : int} (let)}
```

```
\frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau_2} \ (\mathsf{let})
```

The expression let y = 2 in y + y is an int because

» 2 is an int by fiat (and so y is being assigned to a well-typed expression)

```
\frac{\text{\{y:int\}} \vdash y:int}{\{y:int\} \vdash y:int} \frac{\text{\{y:int\}} \vdash y:int}{\{y:int\} \vdash y:int} \text{(intAdd)}}{\{y:int\} \vdash y:int} \text{(let)}
\{\} \vdash \text{let } y = 2 \text{ in } y + y:int}
```

```
\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau_2} \ (\mathsf{let})
```

```
\frac{\mathbf{n} \text{ is an integer literal}}{\Gamma \vdash \mathbf{n} : \mathbf{int}} \text{ (intLit)}
```

The expression let y = 2 in y + y is an int because

» 2 is an int by fiat (and so y is being assigned to a well-typed expression)

```
\{\} \vdash \text{let } y = 2 \text{ in } y + y : \text{int}
```

```
\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let} \ x = e_1 \ \mathsf{in} \ e_2 : \tau_2} \ (\mathsf{let})
```

```
n is an integer literal (intLit)
```

The expression let y = 2 in y + y is an int because

 \gg 2 is an **int** by fiat (and so **y** is being assigned to a well-typed $\Gamma \vdash e_1 : \text{int} \Gamma \vdash e_2 : \text{int} \Gamma \vdash e_2 : \text{int}$ (addInt)

 \gg and, assuming y is an int, y + y is an int because

$$rac{\Gamma dash e_1 : \mathtt{int}}{\Gamma dash e_1 + e_2 : \mathtt{int}} \ (\mathsf{addInt})$$

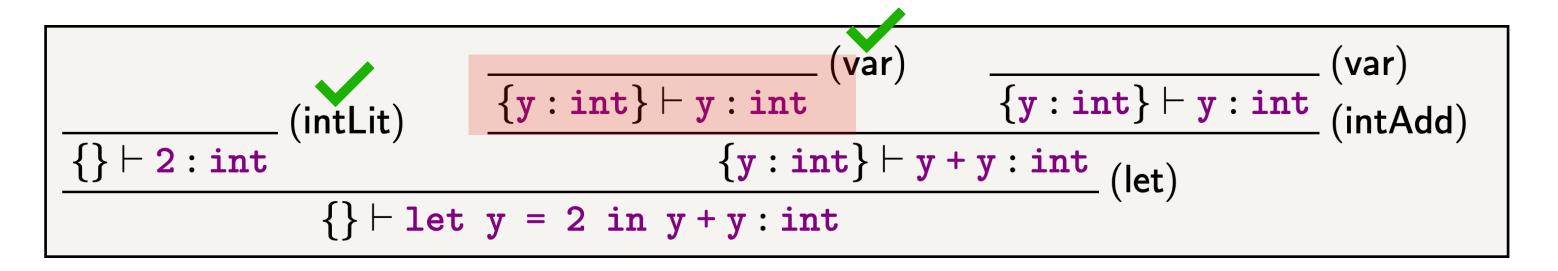
```
 \frac{\{y: int\} \vdash y: int}{\{y: int\} \vdash y: int} (var) - \frac{\{y: int\} \vdash y: int}{\{y: int\} \vdash y: int} (intAdd) 
   \{\} \vdash \text{let } y = 2 \text{ in } y + y : \text{int}
```

```
\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau_2} \ (\mathsf{let})
```

```
n is an integer literal (intLit)
```

- \gg 2 is an **int** by fiat (and so **y** is being assigned to a well-typed $\boxed{\frac{\Gamma \vdash e_1 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}}$ (addInt)
- \gg and, assuming y is an int, y + y is an int because
 - y is an int (by assumption)

$$rac{\Gammadash e_1: ext{int}}{\Gammadash e_1+e_2: ext{int}} \; (ext{addInt})$$



```
\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau_2} \ (\mathsf{let})
```

$$\frac{\mathbf{n} \text{ is an integer literal}}{\Gamma \vdash \mathbf{n} : \mathbf{int}} \text{ (intLit)}$$

- » 2 is an int by fiat (and so y is being assigned to a well-typed)
- \gg and, assuming y is an int, y + y is an int because
 - y is an int (by assumption)

$$rac{\Gammadash e_1: extbf{int}}{\Gammadash e_1+e_2: extbf{int}}$$
 (addInt)

$$rac{(x: au)\in\Gamma}{\Gammadash x: au}$$
 (var)

```
\frac{ \frac{\{y: int\} \vdash y: int\}}{\{y: int\} \vdash y: int\}} (var) }{\{y: int\} \vdash y: int\}} (var) 
\frac{\{y: int\} \vdash y: int\}}{\{y: int\} \vdash y: int\}} (let)
\{\} \vdash let \ y = 2 \ in \ y + y: int}
```

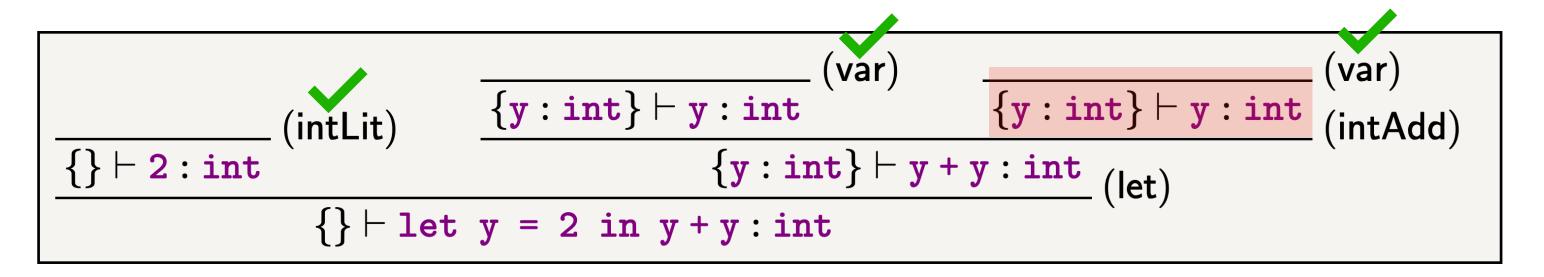
```
\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau_2} \ (\mathsf{let})
```

```
\frac{\mathbf{n} \text{ is an integer literal}}{\Gamma \vdash \mathbf{n} : \mathbf{int}} \text{ (intLit)}
```

- » 2 is an int by fiat (and so y is being assigned to a well-typed
- \gg and, assuming y is an int, y + y is an int because
 - y is an int (by assumption)
 - and so is y (by assumption)

$$\Gamma dash e_1: \mathtt{int} \qquad \Gamma dash e_2: \mathtt{int} \ \Gamma dash e_1 + e_2: \mathtt{int}$$

$$\dfrac{(x: au)\in\Gamma}{\Gammadash x: au}$$
 (var)



```
\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau_2} \ (\mathsf{let})
```

```
\frac{\mathbf{n} \text{ is an integer literal}}{\Gamma \vdash \mathbf{n} : \mathbf{int}} \text{ (intLit)}
```

The expression let y = 2 in y + y is an int because

- » 2 is an int by fiat (and so y is being assigned to a well-typed
- » and, assuming y is an int, y + y is an int because
 - and so is **y** (by assumption)

y is an int (by assumption)

$$rac{\Gammadash e_1: extbf{int}}{\Gammadash e_1+e_2: extbf{int}}$$
 (addInt)

$$rac{(x: au)\in\Gamma}{\Gammadash x: au}$$
 (var)

```
\frac{\{y: \mathtt{int}\} \vdash y: \mathtt{int}}{\{y: \mathtt{int}\} \vdash y: \mathtt{int}} (\mathtt{var}) = \frac{\{y: \mathtt{int}\} \vdash y: \mathtt{int}}{\{y: \mathtt{int}\} \vdash y + y: \mathtt{int}} (\mathtt{int} \mathsf{Add})}
\{\} \vdash let y = 2 in y + y : int
```

```
 \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let} \ x = e_1 \ \mathsf{in} \ e_2 : \tau_2} \ (\mathsf{let})
```

$$\frac{\mathbf{n} \text{ is an integer literal}}{\Gamma \vdash \mathbf{n} : \mathbf{int}} \text{ (intLit)}$$

The expression let y = 2 in y + y is an int because

- $\gg 2$ is an **int** by fiat (and so y is being assigned to a well-typed
- \gg and, assuming y is an int, y + y is an int because
 - y is an int (by assumption)
 - and so is y (by assumption)

$$rac{\Gamma dash e_1 : \mathtt{int}}{\Gamma dash e_1 + e_2 : \mathtt{int}} \ (\mathsf{addInt})$$

$$rac{(x: au)\in\Gamma}{\Gammadash x: au}$$
 (var)

and so integer—adding these two expressions (y and y) yields an int

```
\frac{ \frac{}{\{y: int\} \vdash y: int} (var)}{\{y: int\} \vdash y: int} \frac{\{y: int\} \vdash y: int}{\{y: int\} \vdash y: int} (intAdd)}{\{y: int\} \vdash y: int} (let)
```

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau_2} \ (\mathsf{let})$$

$$\frac{\mathbf{n} \text{ is an integer literal}}{\Gamma \vdash \mathbf{n} : \mathbf{int}} \text{ (intLit)}$$

The expression let y = 2 in y + y is an int because

- » 2 is an int by fiat (and so y is being assigned to a well-typed)
 » and, assuming y is an int, y + y is an int because
 - y is an int (by assumption)
 - and so is y (by assumption)

$$rac{\Gammadash e_1: extbf{int}}{\Gammadash e_1+e_2: extbf{int}} \ (extbf{addInt})$$

$$rac{(x: au)\in\Gamma}{\Gammadash x: au}$$
 (var)

and so integer—adding these two expressions (y and y) yields an int and so assigning y to z in z yields an z

```
\frac{ \frac{}{\{y: int\} \vdash y: int} (var)}{\{y: int\} \vdash y: int} \frac{\{y: int\} \vdash y: int}{\{y: int\} \vdash y: int} (intAdd)}{\{y: int\} \vdash y: int} (let)
```

```
\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau_2} \ (\mathsf{let})
```

 $\frac{\mathbf{n} \text{ is an integer literal}}{\Gamma \vdash \mathbf{n} : \mathbf{int}} \text{ (intLit)}$

The expression let y = 2 in y + y is an int because

- » 2 is an int by fiat (and so y is being assigned to a well-typed |
 » and, assuming y is an int, y + y is an int because
 - y is an int (by assumption)
 - and so is y (by assumption)

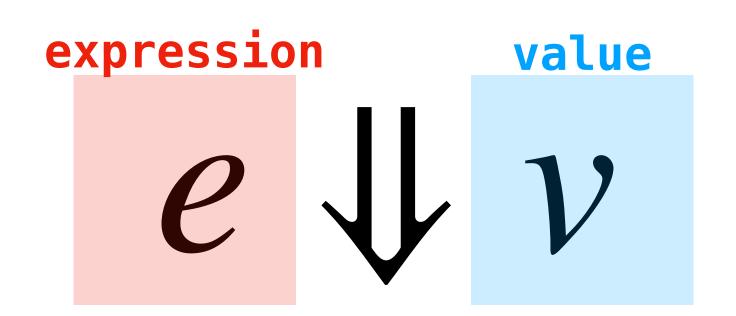
 $rac{\Gammadash e_1: extbf{int}}{\Gammadash e_1+e_2: extbf{int}} \ (extbf{addInt})$

$$rac{(x: au)\in\Gamma}{\Gammadash x: au}$$
 $(extsf{var})$

and so integer—adding these two expressions (y and y) yields an int and so assigning y to 2 in y + y yields an int

And all this works for semantics judgements as well

Recall: Semantic Judgements



A <u>semantic judgment</u> is a compact way of representing the statement:

The expression e evaluates to the value v

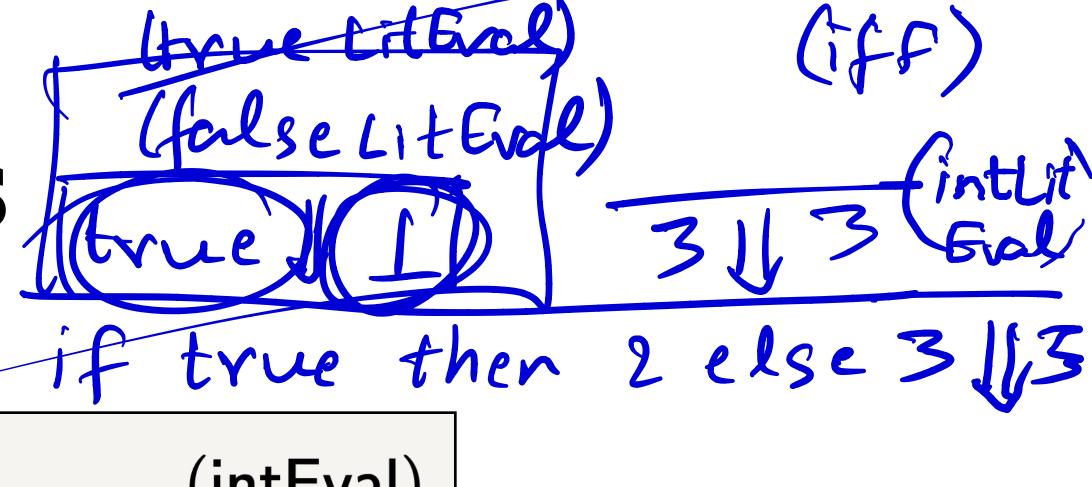
A **semantic rule** is an inference rule with semantic judgments

Recall: Integer Addition Semantic Rule

$$\frac{e_1 \Downarrow v_1}{e_1 + e_2 \Downarrow v} \qquad \frac{v_1 + v_2 = v}{e_1 + e_2 \Downarrow v}$$
 (evalInt)

If e_1 evaluates to the (integer) v_1 and e_2 evaluates to the (integer) v_2 , and $v_1 + v_2 = v$, then $e_1 + e_2$ evaluates to the (integer) v_3

Semantic Derivations



```
\frac{}{\text{true} \downarrow \top} \text{(trueEval)} \quad \frac{}{2 \downarrow 2} \text{(ifEval)} if true then 2 else 3 \downarrow 2
```

We can also write derivations to prove semantic judgments

The principle is the same, except that the judgments are semantic judgments instead of typing judgments

Examples

Pfe:book Pfe: 2 Pfesit Example (Typing) Thit e then et else ent ?

7 Htrue: bool 9+ n: int (intlit)

(trueLit)

(int Lit)

(intLit)

Et Frue: bool

{} Int if true then 2 else 5 : int

Example (Evaluation)

true UT 21/2 (IntEval)

if true then 2 else 5 \$\square\$2\$

Example (Typing)

Cintlit)
$$\frac{\text{Ex:int} + 3:\text{Int}}{\text{Ex:int}} + \frac{\text{Vov}}{\text{Ex:int}} + \frac{\text{Vov}}{\text{IntAdd}} \qquad (\text{intLit})$$

$$\frac{\text{Ex:int}}{\text{Ex:int}} + 3 + x : \text{Int} \qquad \frac{\text{Ex:int}}{\text{Ex:int}} + 4:\text{int} \qquad (< >)$$

$$\{x: \text{int}\} + 3 + x \Leftrightarrow 4: \text{bool}$$

 $let x = 2 in 3 + x \Downarrow 5 \angle$

CletEval

Summary

Derivations are **tree-like proofs** that judgments hold with respect to a collection of inference rules

Derivations are compact mathematical representations of English language arguments

Learning to write derivations takes practice