## Programming Language Theory

Dave Laing

BFPG February 2019

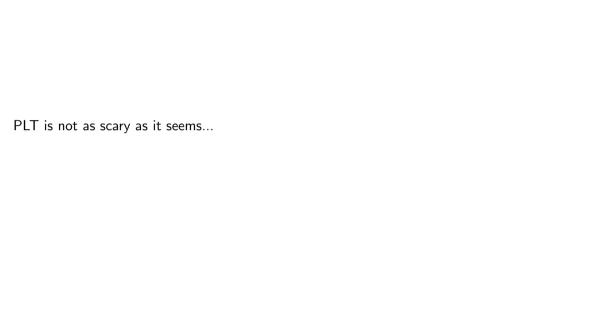


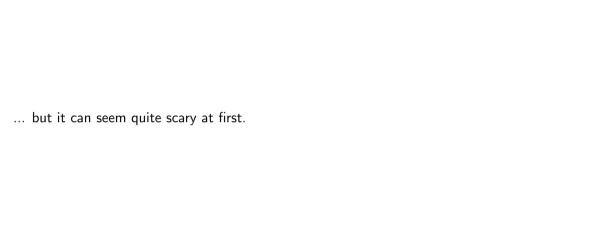
Programming Language Theory (PLT) is handy to know about.	

Useful for understandir languages.	ng how different languages	s work, comparing ther	n, coming up with	ı new



You only need to know about a few things before you can read <i>many</i> mor	e papers.





t :=			
	X	variable	
	$\lambda \times T.t$	abstraction	
	t t	function application	
v :=			
	$\lambda \times T.t$	abstraction	
T :=			
	$T \rightarrow T$	function arrow	

## $\frac{t_2 \longrightarrow t_2'}{v_1 \ t_2 \longrightarrow v_1 \ t_2'}$ (E-APP2) (E-Appabs) $(\lambda x: T.t_1)t_2 \longrightarrow [x \mapsto t_2]t_1$

 $\frac{t_1 \longrightarrow t_1'}{t_1 \ t_2 \longrightarrow t_1' \ t_2}$ 

**Small-step semantics** 

$$\frac{x:T\in\Gamma}{\Gamma\vdash x:T}\tag{T-VAR}$$

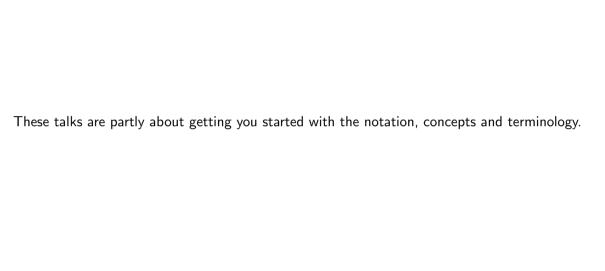
 $\Gamma \vdash t_1 \ t_2 : T_2$  $\Gamma, x: T_1 \vdash T_2$ 

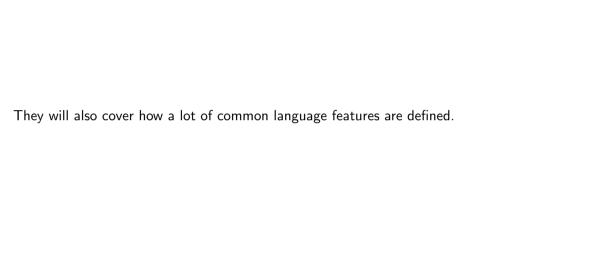
 $\Gamma \vdash (\lambda x:T_1.t):T_1 \rightarrow T_2$ 

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2}{\Gamma \vdash t_1 : T_1 \to T_2} \frac{\Gamma \vdash t_2 : T_1}{\Gamma \vdash t_2 : T_1}$$
 (T-App)

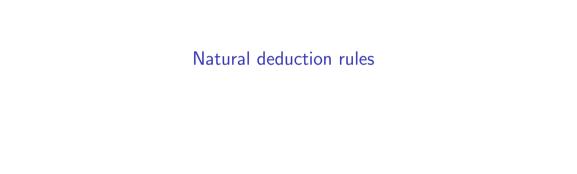
(E-App1)

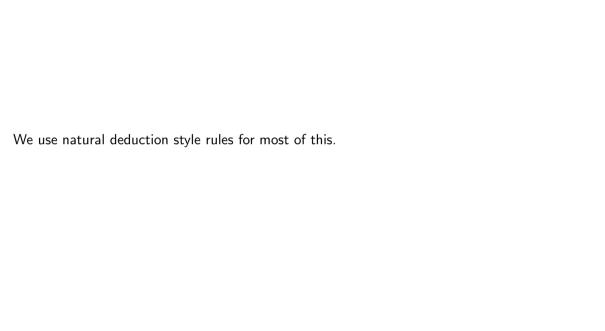
(T-ABS)





It won't cover how to prove these various properties, although "Types and Programming Languages" and "Software Foundations" cover that very well.

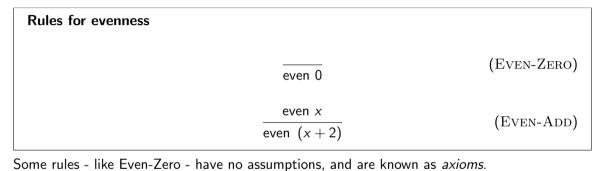




## Assumption<sub>2</sub> ... (Rule-Name)

 $Assumption_1$ 

Conclusion



Some rules - like Even-Zero - have no assumptions, and are known as axioms.

Rules for evenness	
even 0	(EVEN-ZERO)
$\frac{\text{even } x}{\text{even } (x+2)}$	(Even-Add)

Some rules - like Even-Add - are recursive.

or evenness	
even 0	(Even-Zero)
$\frac{\text{even } x}{\text{even } (x+2)}$	(Even-Add)
C	even x

The relations are determined by the union of all of the rules.

even 0	(LVEN-ZERO)
$\frac{\text{even } x}{\text{even } (x+2)}$	(Even-Add)
Usually only one rule will apply at one time, and so the order we organiz matter.	e these rules in doesn't

(EVEN-ZERO)

Rules for evenness

Rules for evenness	

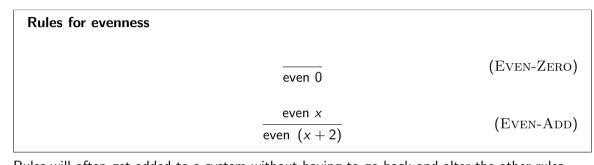
even 0

 $\frac{\text{even } x}{\text{even } (x+2)}$ 

(EVEN-ZERO)

(EVEN-ADD)

We are also dealing with an "open world".



Rules will often get added to a system without having to go back and alter the other rules.

even 0	(Even-Zero)
$\frac{\text{even } x}{\text{even } (x+2)}$	(Even-Add)
$\frac{odd\; x}{even\; (x+1)}$	(Even-Odd)
$\frac{even\; x}{odd\; (x+1)}$	(Odd-Even)

Rules for evenness, now with bonus rules

Let us add some more rules.

Rules for ev	enness, now with bonus rules	
	even 0	(Even-Zero)
	$\frac{\text{even } x}{\text{even } (x+2)}$	(EVEN-ADD)
	$\frac{\operatorname{odd}x}{\operatorname{even}\left(x+1\right)}$	(EVEN-ODD)
	$\frac{even\; x}{odd\; (x+1)}$	(Odd-Even)

Now we could apply these rules in a few different orders to determine even 4.

	$\frac{\text{even } x}{\text{even } (x+2)}$	(Even-Add)
	$\frac{odd\; x}{even\; (x+1)}$	(Even-Odd)
	$\frac{even\; x}{odd\; (x+1)}$	(Odd-Even)
We want the rules to be a paths are finite.	leterministic - the answers through all	of the paths agree, and the

even 0

(Even-Zero)

Rules for evenness, now with bonus rules

$$\frac{P=Q-Q=R}{P=R} \qquad \qquad \text{(Transitivity)}$$
 Sometimes we have rules like Symmetry, which could be applied over and over and spin forever.

P = P

(Reflexivity)

(Symmetry)

	$\overline{P=P}$	(REFLEXIVITY)
	$rac{Q=P}{P=Q}$	(Symmetry)
	$\frac{P=Q\qquad Q=R}{P=R}$	(Transitivity)
Those rules usually exist in the to assist with proofs.	hat form to explain why a system ha	s some desired properties, or

(Reflexivity)

**Equivalence relations** 

## **Equivalence relations**

$$\frac{P=Q-Q=R}{P=R}$$
 (Transitivity)

There will often be a second equivalent set of rules applied that are deterministic and terminating a known as calgorithmic to assist with implementations

P = P

(Reflexivity)

(Symmetry)

- known as algorithmic to assist with implementations.

Equivalence relation	1
----------------------	---

$$\frac{P=Q-Q=R}{P=R} \qquad \qquad \text{(Transitivity)}$$
 This is usually proved by a proof of logical equivalence of the non-deterministic and algorithmic rules sets.

P = P

(Reflexivity)

(Symmetry)

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

> Nothing <|> Nothing

> Nothing <|> Nothing
Nothing

```
> Nothing <|> Nothing
Nothing
```

> Just 1 <|> Nothing

```
> Nothing <|> Nothing
Nothing
> Just 1 <|> Nothing
Just 1
```

```
> Nothing <|> Nothing
Nothing
```

> Just 1 <|> Nothing
Just 1

> Nothing <|> Just 2

```
> Nothing <|> Nothing
Nothing
> Just 1 <|> Nothing
Just 1
> Nothing <|> Just 2
Just 2
```

```
> Nothing <|> Nothing
Nothing
> Just 1 <|> Nothing
Just 1
```

> Nothing <|> Just 2
Just 2

> Just 1 <|> Just 2

```
> Nothing <|> Nothing
Nothing
> Just 1 <|> Nothing
Just 1
```

> Just 1 <|> Just 2

> Nothing <|> Just 2

Just 1

Just 2

```
asum :: (Foldable t, Alternative f) => t (f a) -> f a
asum = foldr (<|>) empty
```

> asum [Just 1, Just 2, Just 3]
Just 1

> asum [Just 1, Just 2, Just 3]
Just 1

> asum [Nothing, Just 2, Just 3]

> asum [Just 1, Just 2, Just 3]
Just 1
> asum [Nothing, Just 2, Just 3]

Just 2

```
> asum [Just 1, Just 2, Just 3]
Just 1
```

> asum [Nothing, Just 2, Just 3]

Just 2
> asum [Nothing Nothing Nothing

> asum [Nothing, Nothing]

```
> asum [Just 1, Just 2, Just 3]
Just 1
```

> asum [Nothing, Just 2, Just 3]
Just 2

> asum [Nothing, Nothing]

Nothing

```
> asum [Just 1, Just 2, Just 3]
Just 1
> asum [Nothing, Just 2, Just 3]
Just 2
> asum [Nothing, Nothing, Nothing]
Nothing
```

> asum []

```
> asum [Just 1, Just 2, Just 3]
Just 1
> asum [Nothing, Just 2, Just 3]
Just 2
> asum [Nothing, Nothing, Nothing]
Nothing
```

> asum []

Nothing

type RuleSet a b = a -> Maybe b

type RuleSet a b = a -> Maybe b
type Rule a b = RuleSet a b -> a -> Maybe b

type RuleSet a b = a -> Maybe b

type Rule a b = RuleSet a b -> a -> Maybe b

mkRuleSet :: [Rule a b] -> RuleSet a b

```
type RuleSet a b = a -> Maybe b
type Rule a b = RuleSet a b -> a -> Maybe b
mkRuleSet :: [Rule a b] -> RuleSet a b
mkRuleSet rules =
  let
    ruleSet a =
      asum .
      fmap (\r -> r ruleSet a) $
      rules
  in
    ruleSet
```

```
evenAddTwo :: Rule Int ()
evenAddTwo e n
| n >= 2 = e (n - 2)
\frac{\text{even } x}{\text{even } (x + 2)}
(EVEN-ADD)
```

otherwise = Nothing

```
evenR :: RuleSet Int ()
evenR = mkRuleSet [evenZero, evenAddTwo]
```

```
evenOdd :: RuleSet Int () -> Rule Int ()
evenOdd o _ n
\mid n >= 1 = o (n - 1)
\frac{\text{odd } x}{\text{even } (x+1)}
(EVEN-ODD)
```

otherwise = Nothing

```
evenR, oddR :: RuleSet Int ()
evenR = mkRuleSet [evenZero, evenOdd oddR]
oddR = mkRuleSet [oddEven evenR]
```

```
evenR, oddR :: RuleSet Int ()
evenR = mkRuleSet [evenZero, evenAddTwo, evenOdd oddR]
oddR = mkRuleSet [oddEven evenR]
```

Nothing

## Nothing

> evenR 4

```
> evenR 3
Nothing
```

Just ()

```
> evenR 3
Nothing
```

Just ()

> oddR 3

```
> evenR 3
Nothing
```

Just ()

> oddR 3

Just ()

```
> evenR 3
Nothing
```

Just ()
> oddR 3

Just ()

> oddR 4

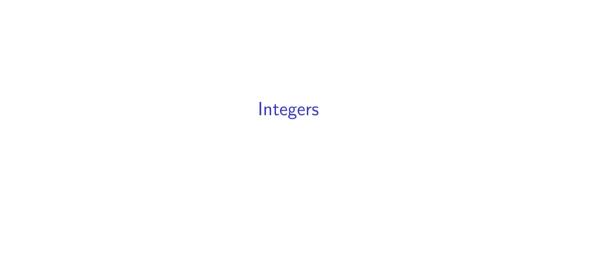
```
> evenR 3
Nothing
```

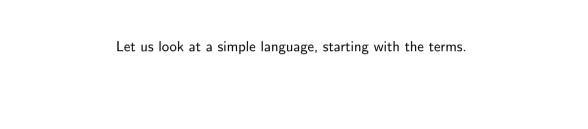
Just ()
> oddR 3

Just ()

> oddR 4

 ${\tt Nothing}$ 





## **Terms**

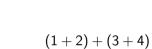
 $\langle \mathsf{int} \rangle$ constant integer addition t + t

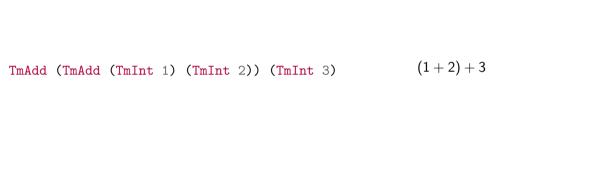
These are the pieces of the language and the ways those pieces can be combined.

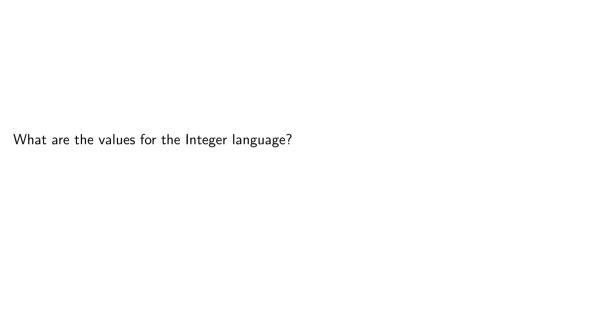
## **Terms**

 $\langle int \rangle$ constant integer addition t + t

This is the abstract syntax of our language.







#### By definitions:

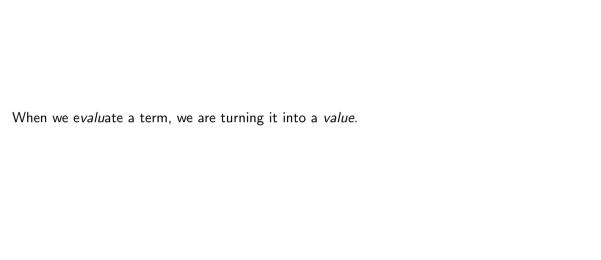
:=

(int) constant integer

We define the values in terms of the syntax of the language.

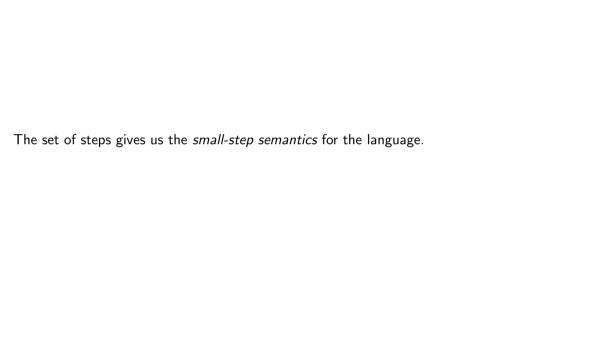
By rules:  $\frac{}{\mathsf{value} \; \langle \mathsf{int} \rangle} \tag{V-Int}$ 

Having the rules can help with the implementation.



Values are specified as part of the <i>syntax</i> of a languge.	





The steps ar	re specified as	a binary relatio	on $t_1 \longrightarrow t_2$ .		

The relation	$t_1 \longrightarrow t_2$	indicates th	at the term	$t_1$ can step	to $t_2$ .	

-

### Small-step semantics (partial)

 $\langle \mathsf{int}_1 \rangle + \langle \mathsf{int}_2 \rangle \longrightarrow \langle \mathsf{int}_1 + \mathsf{int}_2 \rangle$ 

(E-AddInt)

E-AddInt does the actual addition.

## **Small-step semantics (partial)**

$$\frac{\text{value }t_1 \qquad t_2 \longrightarrow t_2'}{t_1 + t_2 \longrightarrow t_1 + t_2'} \tag{E-Add1}$$
 E-Add2 control the order in which the steps are applied to get to the point where

 $\frac{t_1 \longrightarrow t_1'}{t_1 + t_2 \longrightarrow t_1' + t_2}$ 

(E-ADD1)

E-Add1 and E-Add2 control the order in which the steps are applied to get to the point where E-AddInt applies.

## Small-step semantics (partial)

$$egin{aligned} rac{t_1 \longrightarrow t_1'}{t_1 + t_2 \longrightarrow t_1' + t_2} \ rac{t_2 \longrightarrow t_2'}{v_1 + t_2 \longrightarrow v_1 + t_2'} \end{aligned}$$

(E-Add2)

(E-ADD1)

We can use conventions of notation to simplify references to terms that should be values.

# **Small-step semantics**

# $rac{{t_1 \longrightarrow t_1}'}{t_1 + t_2 \longrightarrow {t_1}' + t_2}$

$$t_2 \longrightarrow {t_2}'$$

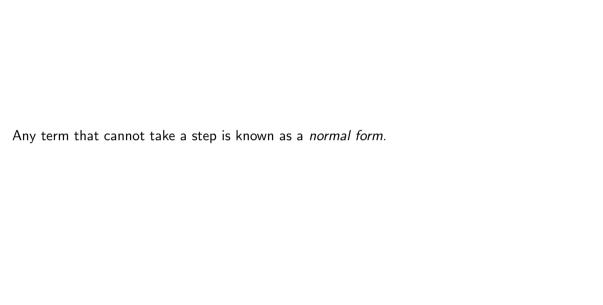
(E-ADD1)

(E-ADD2)

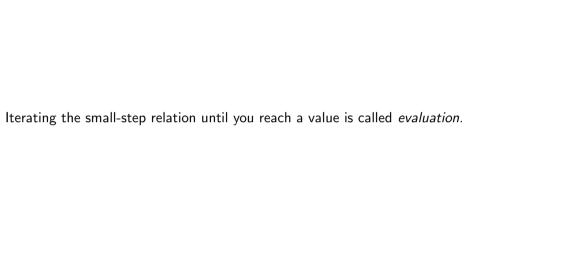
(E-AddInt)

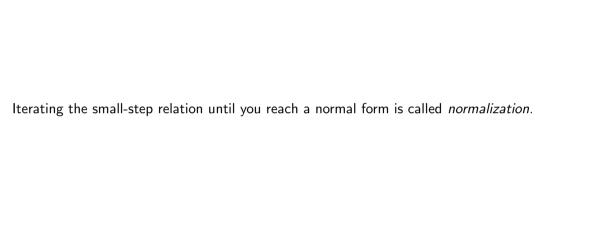
$$\frac{t_2 \longrightarrow t_2'}{v_1 + t_2 \longrightarrow v_1 + t_2'}$$

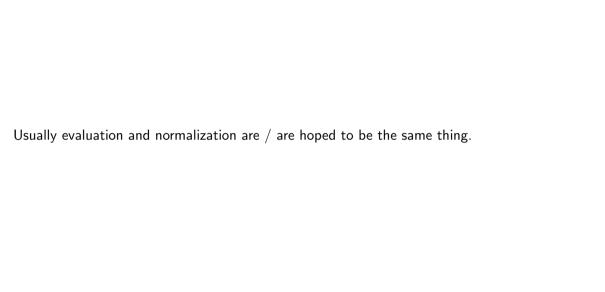
 $\langle \mathsf{int}_1 \rangle + \langle \mathsf{int}_2 \rangle \longrightarrow \langle \mathsf{int}_1 + \mathsf{int}_2 \rangle$ 

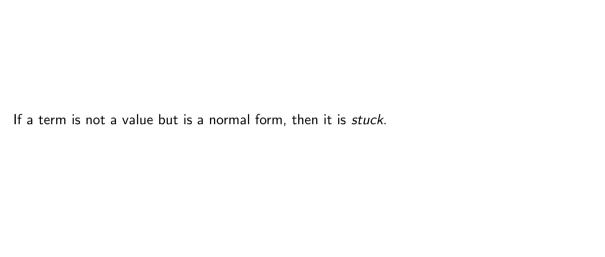


Values cannot take a step by definition and so are always normal forms.	









A language can be <i>normalizing</i> : there is an evaluation order that means that finite-sized t will always evaluate in finite time.	erms

A language can be <i>strongly normalizing</i> : for any evaluation order, finite-sized terms will alw evaluate in finite time.	vays

ne relationship between values and normal forms is a relationship between syntax and mantics.	

Aside: We can define evaluation in terms of a big-step relation:

(BIG-VALUE)  $v \Rightarrow v$ 

$$\frac{1}{v \Rightarrow v}$$

$$\longrightarrow t' \qquad t' \Rightarrow v$$
 (Big-Step)

Let us evaluate (1+2) + (3+4).

$$\cfrac{1+2 \longrightarrow 3}{\left(1+2\right)+\left(3+4\right) \longrightarrow 3+\left(3+4\right)} \text{E-Add1}$$

-----F-AddInt

Then:

$$\frac{3+4 \longrightarrow 7}{3+4 \longrightarrow 7} E-AddInt$$

$$\frac{3+4 \longrightarrow 7}{3+4 \longrightarrow 3+7} E-Add2$$

 $3+7 \longrightarrow 10$ 

Finally: ———— E-AddInt

$$\frac{1+2 \longrightarrow 3}{\left(\boxed{1+2}\right)+\left(3+4\right) \longrightarrow 3+\left(3+4\right)} \text{E-Add1}$$

----- E-AddInt

E-Add2

Then:

$$\frac{-3+4 \longrightarrow 7}{-3+4} = -3 = -3$$

Finally:

$$3 + (3+4) \longrightarrow 3+7$$

F-AddInt

First:

$$\begin{array}{c|c}
\hline
1+2 \longrightarrow 3 \\
\hline
\left(1+2\right) + \left(3+4\right) \longrightarrow 3 + \left(3+4\right)
\end{array}$$
E-AddInt

E-Add2

Then:

$$\frac{3+4 \longrightarrow 7}{2+(3+4)} = 2+7$$

 $3 + (3+4) \longrightarrow 3+7$ Finally:

First:

Then:

$$\frac{3+4 \longrightarrow 7}{3+4 \longrightarrow 7} E-AddInt$$

$$\frac{3+4 \longrightarrow 7}{3+4 \longrightarrow 3+7} E-Add2$$

Finally:

E-Add2

Then:

First:

E-AddInt
$$\frac{1+2 \longrightarrow 3}{\left(1+2\right)+\left(3+4\right) \longrightarrow 3+\left(3+4\right)} = E-Add1$$

Then:

$$\frac{3+4 \longrightarrow 7}{3+4 \longrightarrow 7} \text{E-AddInt}$$

$$\frac{3+4 \longrightarrow 7}{3+(3+4) \longrightarrow 3+7}$$

$$\frac{1+2 \longrightarrow 3}{\left(1+2\right)+\left(3+4\right) \longrightarrow \left(3+4\right)} \text{E-Add1}$$

----- E-AddInt

Then:

First:

$$\frac{1+2 \longrightarrow 3}{\left(1+2\right)+\left(3+4\right) \longrightarrow 3+\left(3+4\right)} \text{E-Add1}$$

E-AddInt

Then:

Finally:

$$\frac{1+2 \longrightarrow 3}{\left(1+2\right)+\left(3+4\right) \longrightarrow 3+\left(3+4\right)} \text{E-Add1}$$

 $3+7 \longrightarrow 10$ 

----- E-AddInt

E-Add2

F-AddInt

Then:

$$\frac{1+2 \longrightarrow 3}{\left(1+2\right)+\left(3+4\right) \longrightarrow 3+\left(3+4\right)} \text{E-Add1}$$

----- E-AddInt

Then:

$$\frac{3+4}{3+4} \longrightarrow 7$$
E-AddInt
$$\frac{3+4}{3+4} \longrightarrow 7$$
E-Add2

 $3 + (3+4) \longrightarrow 3+7$ 

Finally:

$$\frac{1+2 \longrightarrow 3}{\left(1+2\right)+\left(3+4\right) \longrightarrow 3+\left(3+4\right)} \text{E-Add1}$$

E-AddInt

Then:

$$3+4 \longrightarrow$$

 $3 + (3+4) \longrightarrow 3+7$ 

$$3+\left(\boxed{3+4}\right)$$

$$3 + (3+4)$$
 Finally:

$$\frac{1+2 \longrightarrow 3}{\left(1+2\right)+\left(3+4\right) \longrightarrow 3+\left(3+4\right)} \text{E-Add1}$$

F-AddInt

Then:

$$\frac{3+4 \longrightarrow 7}{\text{E-AddInt}}$$
E-Add2

Finally:

$$3 + \left(3+4\right) \longrightarrow 3 + 7$$

First:

$$\frac{1+2 \longrightarrow 3}{\left(1+2\right)+\left(3+4\right) \longrightarrow 3+\left(3+4\right)} \text{E-Add1}$$

F-AddInt

Then:

$$\frac{3+4 \longrightarrow 7}{\text{E-AddInt}}$$
E-Add2

$$\frac{3+4 \longrightarrow 7}{3+(3+4) \longrightarrow 3+7}$$
E-

 $3+7 \longrightarrow 10$ 

Finally:

$$\frac{1+2 \longrightarrow 3}{\left(1+2\right)+\left(3+4\right) \longrightarrow 3+\left(3+4\right)} \text{E-Add1}$$

F-AddInt

Finally:

Then:

----- E-AddInt

 $3+7 \longrightarrow 10$ 

F-AddInt

$$\frac{1+2 \longrightarrow 3}{\left(1+2\right)+\left(3+4\right) \longrightarrow 3+\left(3+4\right)} \text{E-Add1}$$

E-AddInt

Then:

$$\frac{3+4 \longrightarrow 7}{3+\left(3+4\right) \longrightarrow 3+7} = 3+7$$

Finally: 
$$\frac{3 + (3 + 4)}{} \longrightarrow \frac{3 + 7}{}$$

$$\frac{1+2 \longrightarrow 3}{\left(1+2\right)+\left(3+4\right) \longrightarrow 3+\left(3+4\right)} \text{E-Add1}$$

F-AddInt

E-Add2

Then:

$$\frac{3+4 \longrightarrow 7}{3+(3+4) \longrightarrow 3+7}$$

$$3 + (3+4) \longrightarrow 3+7$$

First:

$$\frac{1+2 \longrightarrow 3}{\left(1+2\right)+\left(3+4\right) \longrightarrow 3+\left(3+4\right)} \text{E-Add1}$$

F-AddInt

Then:

$$\frac{3+4 \longrightarrow 7}{3+4 \longrightarrow 7} \text{E-AddInt}$$

$$\frac{3+4 \longrightarrow 7}{3+(3+4) \longrightarrow 3+7}$$

Finally:

$$\frac{1+2 \longrightarrow 3}{\left(1+2\right)+\left(3+4\right) \longrightarrow 3+\left(3+4\right)} \text{E-Add1}$$

----- F-AddInt

E-Add2

E-AddInt

Then:

```
vInt :: Rule Term ()
vInt _ (TmInt _) =
    Just ()
intValue _ _ =
    Nothing
(V-Int)
```

```
valueR :: RuleSet Term ()
valueR =
  mkRuleSet [vInt]
```

```
> let tm = TmAdd (TmAdd (TmInt 1) (TmInt 2)) (TmAdd (TmInt 3) (TmInt 4))
```

```
> let tm = TmAdd (TmAdd (TmInt 1) (TmInt 2)) (TmAdd (TmInt 3) (TmInt 4))
> valueR tm
```

```
> let tm = TmAdd (TmAdd (TmInt 1) (TmInt 2)) (TmAdd (TmInt 3) (TmInt 4))
> valueR tm
```

Nothing

```
> let tm = TmAdd (TmAdd (TmInt 1) (TmInt 2)) (TmAdd (TmInt 3) (TmInt 4))
> valueR tm
Nothing
```

> valueR (TmInt 10)

```
> let tm = TmAdd (TmAdd (TmInt 1) (TmInt 2)) (TmAdd (TmInt 3) (TmInt 4))
> valueR tm
Nothing
> valueR (TmInt 10)
Just ()
```

```
eAdd1 :: Rule Term Term
```

 $eAdd1 _ =$ Nothing

tm1' <- step tm1

eAdd1 step (TmAdd tm1 tm2) = do

pure \$ TmAdd tm1' tm2

 $\frac{t_1 \longrightarrow t_1'}{t_1 + t_2 \longrightarrow t_1' + t_2} \quad \text{(E-Add1)}$ 

```
eAdd2 :: RuleSet Term () -> Rule Term Term
eAdd2 value step (TmAdd tm1 tm2) = do
```

 $\frac{t_2 \longrightarrow t_2'}{v_1 + t_2 \longrightarrow v_1 + t_2'} \quad \text{(E-Add2)}$ 

<- value tm1

tm2' <- step tm2

pure \$ TmAdd tm1 tm2'

eAdd2 \_ \_ = Nothing

```
eAdd2 :: Rule Term Term
```

 $eAdd2 _ =$ Nothing

eAdd2 step (TmAdd tm1@(TmInt \_) tm2) = do

pure \$ TmAdd tm1 tm2'

 $\frac{t_2 \longrightarrow t_2'}{v_1 + t_2 \longrightarrow v_1 + t_2'} \quad \text{(E-Add2)}$ 

Nothing

```
stepR :: RuleSet Term Term
stepR =
  mkRuleSet [eAdd1, eAdd2, eAddInt]
```

```
> let tm = TmAdd (TmAdd (TmInt 1) (TmInt 2)) (TmAdd (TmInt 3) (TmInt 4))
```

```
> let tm = TmAdd (TmAdd (TmInt 1) (TmInt 2)) (TmAdd (TmInt 3) (TmInt 4))
```

> stepR tm

```
> let tm = TmAdd (TmAdd (TmInt 1) (TmInt 2)) (TmAdd (TmInt 3) (TmInt 4))
> stepR tm
```

Just (TmAdd (TmInt 3) (TmAdd (TmInt 3) (TmInt 4)))

```
> let tm = TmAdd (TmAdd (TmInt 1) (TmInt 2)) (TmAdd (TmInt 3) (TmInt 4))
> stepR tm
```

Just (TmAdd (TmInt 3) (TmAdd (TmInt 3) (TmInt 4)))

> stepR >=> stepR \$ tm

```
> let tm = TmAdd (TmAdd (TmInt 1) (TmInt 2)) (TmAdd (TmInt 3) (TmInt 4))
> stepR tm
Just (TmAdd (TmInt 3) (TmAdd (TmInt 3) (TmInt 4)))
```

> stepR >=> stepR \$ tm

Just (TmAdd (TmInt 3) (TmInt 7))

```
> let tm = TmAdd (TmAdd (TmInt 1) (TmInt 2)) (TmAdd (TmInt 3) (TmInt 4))
> stepR tm
Just (TmAdd (TmInt 3) (TmAdd (TmInt 3) (TmInt 4)))
```

> stepR >=> stepR \$ tm

```
Just (TmAdd (TmInt 3) (TmInt 7))
```

> stepR >=> stepR >=> stepR \$ tm

```
> let tm = TmAdd (TmAdd (TmInt 1) (TmInt 2)) (TmAdd (TmInt 3) (TmInt 4))
> stepR tm
Just (TmAdd (TmInt 3) (TmAdd (TmInt 3) (TmInt 4)))
```

> stepR >=> stepR >=> stepR \$ tm

Just (TmAdd (TmInt 3) (TmInt 7))

> stepR >=> stepR \$ tm

Just (TmInt 10)

```
> let tm = TmAdd (TmAdd (TmInt 1) (TmInt 2)) (TmAdd (TmInt 3) (TmInt 4))
> stepR tm
Just (TmAdd (TmInt 3) (TmAdd (TmInt 3) (TmInt 4)))
```

Just (TmAdd (TmInt 3) (TmInt 7))
> stepR >=> stepR \$ tm

> stepR >=> stepR \$ tm

Just (TmInt 10)

> stepR >=> stepR >=> stepR >=> stepR \$ tm

```
> let tm = TmAdd (TmAdd (TmInt 1) (TmInt 2)) (TmAdd (TmInt 3) (TmInt 4))
> stepR tm
Just (TmAdd (TmInt 3) (TmAdd (TmInt 3) (TmInt 4)))
```

Just (TmAdd (TmInt 3) (TmInt 7))
> stepR >=> stepR >=> stepR \$ tm

> stepR >=> stepR \$ tm

Just (TmInt 10)

> stepR >=> stepR >=> stepR >=> stepR \$ tm

Nothing

```
iterR :: RuleSet a a -> a -> a
iterR r x = case r x of
  Nothing -> x
  Just x' -> iterR r x'
```

eval :: Term -> Term
eval =

iterR stepR

```
> let tm = TmAdd (TmAdd (TmInt 1) (TmInt 2)) (TmAdd (TmInt 3) (TmInt 4))
```

```
> let tm = TmAdd (TmAdd (TmInt 1) (TmInt 2)) (TmAdd (TmInt 3) (TmInt 4))
> eval tm
```

```
> let tm = TmAdd (TmAdd (TmInt 1) (TmInt 2)) (TmAdd (TmInt 3) (TmInt 4))
> eval tm
Just (TmInt 10)
```

 $orall \ t \ \Big( \ \mathsf{value} \ t \ \lor \ \exists t' \ (t \longrightarrow t') \ \Big)$ 

For all terms, the term is either a value or is able to step.

 $\forall t \ ( \text{ value } t \ \lor \ \exists t' (t \longrightarrow t') \ )$ 

For all terms, the term is either a value or is able to step.

 $orall \, m{t} \, \Big( \, \mathsf{value} \, \, t \, \, \lor \, \, \exists t' \, (t \longrightarrow t') \, \Big)$ 

For all terms, the term is either a value or is able to step.

 $\forall \ t \ \Big( \ \mathsf{value} \ t \ \bigvee \ \exists t' (t \longrightarrow t') \ \Big)$ 

For all terms, the term is either a value or is able to step.

### Terms are not stuck

For all terms, the term is either a value or is able to step.

### Terms are not stuck

 $\forall t \left( \text{ value } t \lor \left| \exists t'(t \longrightarrow t') \right| \right)$ 

For all terms, the term is either a value or is able to step.

```
genTerm :: Gen Term
genTerm =
   Gen.recursive Gen.choice
   [ TmInt <$> Gen.int (Range.linear 0 10) ]
   [ Gen.subterm2 genTerm genTerm TmAdd ]
```

```
> Gen.printTree . Gen.small $ genTerm
TmAdd (TmInt 1) (TmInt 1)
|-TmInt 0
|-TmInt 1
```

I-TmTnt 1

I-TmInt O

|-TmAdd (TmInt 0) (TmInt 1)

|-TmAdd (TmInt 1) (TmInt 0)

|-TmAdd (TmInt 0) (TmInt 0)

exactlyOne ga r1 r2 = property \$ do

isJust (r1 a) /== isJust (r2 a)

a <- forAll ga

```
intRulesValueOrStep :: Property
intRulesExactlyOne =
  exactlyOne genTerm valueR stepR
```

>	check	\$ intRulesValueOrStep			

> check \$ intRulesValueOrStep
 <interactive> passed 100 tests.
True

### Booleans

## Terms and values $\begin{array}{cccc} t & := & \\ & \text{false} \\ & \text{true} \end{array}$

constant false

constant true disjunction

false value true value

t or t

false

true

### Small-step semantics (eager)

 $\frac{t_1 \longrightarrow t_1'}{t_1 \text{ or } t_2 \longrightarrow t_1' \text{ or } t_2} \quad \text{(E-OR1)}$ 

 $\frac{}{\text{true or true}} \rightarrow \text{true}$  (E-OrTrueTrue)

 $\frac{}{\text{false or false} \longrightarrow \text{false}} \text{(E-OrFalseFalse)}$ 

 $\frac{t_2 \longrightarrow t_2'}{v_1 \text{ or } t_2 \longrightarrow v_1 \text{ or } t_2'}$  (E-OR2)  $\overline{\text{true or false} \longrightarrow \text{true}}$  (E-ORTRUEFALSE)

 $\frac{}{\text{false or true}} \rightarrow \text{true} \left(\text{E-OrFalseTrue}\right)$ 

# Small-step semantics (lazy) $\frac{t_1 \longrightarrow t_1'}{t_1 \text{ or } t_2 \longrightarrow t_1' \text{ or } t_2} \tag{E-Or1}$ $\frac{\text{false or } t_2 \longrightarrow t_2}{\text{false or } t_2 \longrightarrow t_2}$

true or  $t_2 \longrightarrow \text{true}$ 

(E-OrTrue)

```
data Term =
    TmFalse
    | TmTrue
    | TmOr Term Term
    deriving (Eq, Ord, Show)
```

```
vFalse :: Rule Term ()
vFalse _ TmFalse =
   Just ()
vFalse _ _ =
   Nothing
(V-FALSE)
```

```
vTrue :: Rule Term ()
vTrue _ TmTrue =
   Just ()
vTrue _ _ =
   Nothing
(V-TRUE)
```

```
valueR :: RuleSet Term ()
valueR =
  mkRuleSet [vFalse, vTrue]
```

```
eOr1 :: Rule Term Term
eOr1 step (TmOr tm1 tm2) = do
 tm1' <- step tm1
```

pure \$ TmOr tm1' tm2

eOr1 \_ = Nothing

 $\frac{t_1 \longrightarrow t_1'}{t_1 \text{ or } t_2 \longrightarrow t_1' \text{ or } t_2} \quad \text{(E-OR1)}$ 

```
eOr2 :: RuleSet Term () -> Rule Term Term
```

e0r2 value step (Tm0r tm1 tm2) = do<- value tm1

$$1 \text{ tm2}) = do$$

 $\frac{t_2 \longrightarrow t_2'}{v_1 \text{ or } t_2 \longrightarrow v_1 \text{ or } t_2'} \quad \text{(E-OR2)}$ 

pure \$ TmOr tm1 tm2'

eOr2 \_ \_ = Nothing

- tm2' <- step tm2

Nothing

```
\verb|stepEagerR|:: RuleSet Term| Term|
```

```
stepEagerR =
  mkRuleSet [ eOr1, eOr2
```

, eOrFalseFalse, eOrFalseTrue, eOrTrueFalse, eOrTrueTrue

Nothing

```
eOrTrue :: Rule Term Term
eOrTrue _ (TmOr TmTrue _) =
  Just TmTrue
```

eOrTrue \_ =
Nothing

 $rac{}{\mathsf{true}\;\mathsf{or}\;t_2\longrightarrow\mathsf{true}}\,(\mathrm{E} ext{-}\mathrm{OrTrue})$ 

### stepLazyR :: RuleSet Term Term stepLazyR =

mkRuleSet [eOr1, eOrFalse, eOrTrue]

```
> let tm = TmOr (TmOr TmFalse TmTrue) (TmOr TmFalse TmTrue)
```

- > let tm = TmOr (TmOr TmFalse TmTrue) (TmOr TmFalse TmTrue)
- > stepEagerR tm

```
> let tm = TmOr (TmOr TmFalse TmTrue) (TmOr TmFalse TmTrue)
```

> stepEagerR tm

Just (TmOr TmTrue (TmOr TmFalse TmTrue))

```
> let tm = TmOr (TmOr TmFalse TmTrue) (TmOr TmFalse TmTrue)
```

Just (TmOr TmTrue (TmOr TmFalse TmTrue))

> stepEagerR >=> stepEagerR \$ tm

> stepEagerR tm

- > let tm = TmOr (TmOr TmFalse TmTrue) (TmOr TmFalse TmTrue)
- Just (TmOr TmTrue (TmOr TmFalse TmTrue))
- > stepEagerR >=> stepEagerR \$ tm

Just (TmOr TmTrue TmTrue)

> stepEagerR tm

```
> let tm = TmOr (TmOr TmFalse TmTrue) (TmOr TmFalse TmTrue)
> stepEagerR tm
```

> stepEagerR >=> stepEagerR \$ tm

Just (TmOr TmTrue (TmOr TmFalse TmTrue))

Just (TmOr TmTrue TmTrue)

> stepEagerR >=> stepEagerR >=> stepEagerR \$ tm

```
> let tm = TmOr (TmOr TmFalse TmTrue) (TmOr TmFalse TmTrue)
> stepEagerR tm
Just (TmOr TmTrue (TmOr TmFalse TmTrue))
> stepEagerR >=> stepEagerR $ tm
```

> stepEagerR >=> stepEagerR >=> stepEagerR \$ tm

Just TmTrue

Just (TmOr TmTrue TmTrue)

```
> let tm = TmOr (TmOr TmFalse TmTrue) (TmOr TmFalse TmTrue)
> stepEagerR tm
Just (TmOr TmTrue (TmOr TmFalse TmTrue))
```

> stepEagerR >=> stepEagerR \$ tm

Just (TmOr TmTrue TmTrue)

> stepEagerR >=> stepEagerR >=> stepEagerR \$ tm

Just TmTrue

> stepEagerR >=> stepEagerR >=> stepEagerR >=> stepEagerR \$ tm

```
> let tm = TmOr (TmOr TmFalse TmTrue) (TmOr TmFalse TmTrue)
> stepEagerR tm
Just (TmOr TmTrue (TmOr TmFalse TmTrue))
> stepEagerR >=> stepEagerR $ tm
Just (TmOr TmTrue TmTrue)
> stepEagerR >=> stepEagerR >=> stepEagerR $ tm
```

#### Just TmTrue

> stepEagerR >=> stepEagerR >=> stepEagerR >=> stepEagerR \$ tm

### Nothing

```
> let tm = TmOr (TmOr TmFalse TmTrue) (TmOr TmFalse TmTrue)
```

```
> let tm = TmOr (TmOr TmFalse TmTrue) (TmOr TmFalse TmTrue)
```

> stepLazyR tm

```
> let tm = TmOr (TmOr TmFalse TmTrue) (TmOr TmFalse TmTrue)
```

> stepLazyR tm

Just (TmOr TmTrue (TmOr TmFalse TmTrue))

```
> let tm = TmOr (TmOr TmFalse TmTrue) (TmOr TmFalse TmTrue)
```

Just (TmOr TmTrue (TmOr TmFalse TmTrue))

```
> stepLazyR >=> stepLazyR $ tm
```

> stepLazyR tm

```
> let tm = TmOr (TmOr TmFalse TmTrue) (TmOr TmFalse TmTrue)
> stepLazyR tm
```

Just (TmOr TmTrue (TmOr TmFalse TmTrue))

> stepLazyR >=> stepLazyR \$ tm

Just TmTrue

```
> let tm = TmOr (TmOr TmFalse TmTrue) (TmOr TmFalse TmTrue)
> stepLazyR tm
```

Just (TmOr TmTrue (TmOr TmFalse TmTrue))

> stepLazyR >=> stepLazyR \$ tm

Just TmTrue

> stepLazyR >=> stepLazyR >=> stepLazyR \$ tm

```
> let tm = TmOr (TmOr TmFalse TmTrue) (TmOr TmFalse TmTrue)
> stepLazyR tm
```

Just (TmOr TmTrue (TmOr TmFalse TmTrue))

> stepLazyR >=> stepLazyR \$ tm

Just TmTrue

> stepLazyR >=> stepLazyR >=> stepLazyR \$ tm

Nothing

For all terms, if the term can take a step then

any other step we could take from that same initial term will have the same result .

 $\forall t \ (\exists t'(t \longrightarrow t') \longrightarrow (\exists t''(t \longrightarrow t'') \longrightarrow t' = t''))$ 

For all terms, if the term can take a step

then any other step we could take from that same initial term will have the same result .

For all terms, if

$$orall \; m{t} \; \Big( \; \exists t' \, (t \longrightarrow t') \; \longrightarrow \; \Big( \; \exists t'' \, (t \longrightarrow t'') \; \longrightarrow \; t' = t'' \; \Big) \Big)$$

the term can take a step then any other step we could take from that same initial term will have the same result .

the term can take a step

For all terms, if

then any other step we could take from that same initial term  $% \left( 1\right) =\left( 1\right) +\left( 1\right) =\left( 1\right) +\left( 1\right) +\left( 1\right) =\left( 1\right) +\left( 1$ 

 $orall \ t \ \Big( \ \exists t' \, (t \longrightarrow t') \ igodots \Big( \ \exists t'' \, (t \longrightarrow t'') \ \longrightarrow \ t' = t'' \ \Big) \Big)$ 

For all terms, if

the term can take a step

then

any other step we could take from that same initial term will have the same result .

the term can take a step

For all terms, if

any other step we could take from that same initial term will have the same result .

For all terms, if

$$\forall t \left( \exists t'(t \longrightarrow t') \longrightarrow \left( \exists t''(t \longrightarrow t'') \longrightarrow \boxed{t' = t''} \right) \right)$$

then the term can take a step then any other step we could take from that same initial term will have the same result .

```
genTerm :: Gen Term
genTerm =
   Gen.recursive Gen.choice
   [ pure TmFalse, pure TmTrue ]
   [ Gen.subterm2 genTerm genTerm TmOr ]
```

a1 === a2

pure (a, mkRuleSet rs1 a, mkRuleSet rs2 a)

boolRulesEagerDeterminstic :: Property boolRulesEagerDeterminstic = deterministic genTerm evalRulesEager

boolRulesLazyDeterminstic :: Property boolRulesLazyDeterminstic =

deterministic genTerm evalRulesLazy

```
> check $ boolRulesEagerDeterminstic
```

> check \$ boolRulesEagerDeterminstic
 <interactive> passed 100 tests.
True

```
> check $ boolRulesEagerDeterminstic
  <interactive> passed 100 tests.
True
```

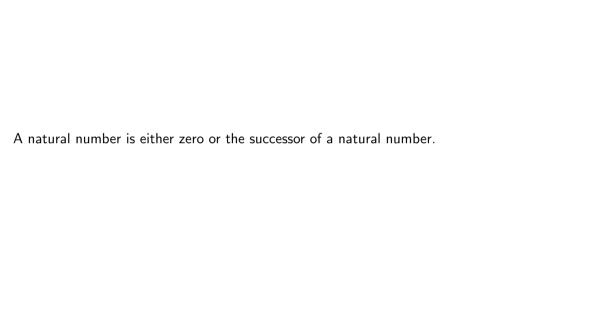
> check \$ boolRulesLazyDeterminstic

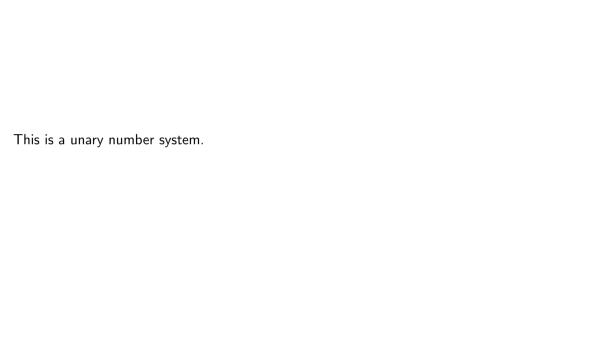
```
> check $ boolRulesEagerDeterminstic
  <interactive> passed 100 tests.
True
```

> check \$ boolRulesLazyDeterminstic
 <interactive> passed 100 tests.

True







# 

succ V

zero value

successor value

# Terms and values (lazy) :=constant zero succ tpred t

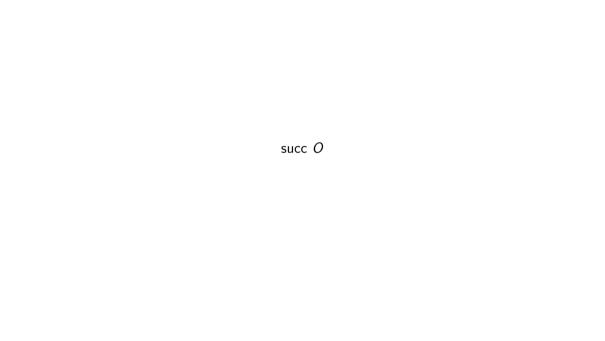
succ t

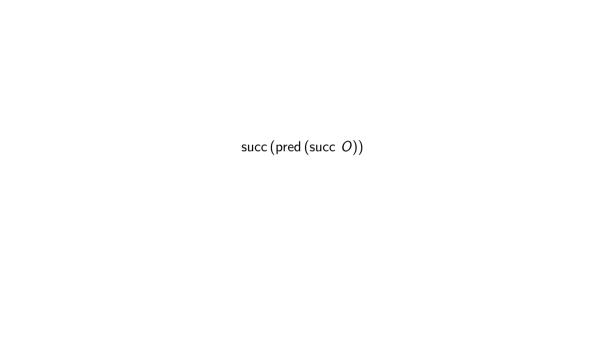
successor

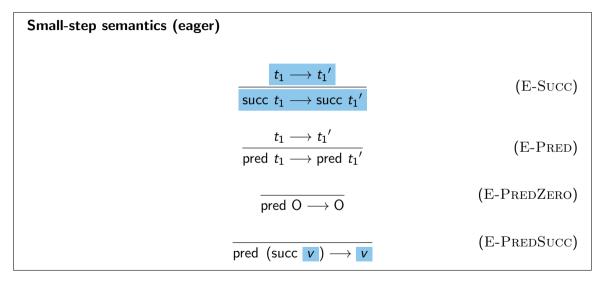
predecessor

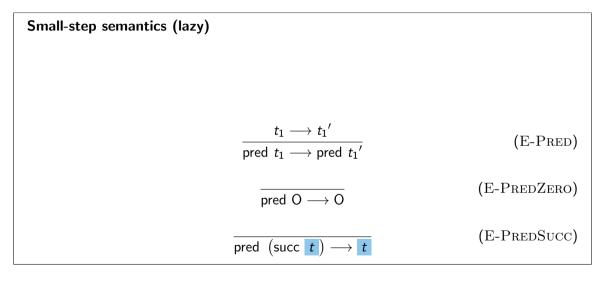
zero value

successor value









These are both versions of the number 1.

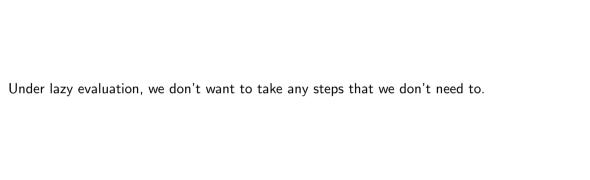
With eager evaluation:

succ O

With lazy evaluation:

succ(pred(succ O))

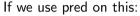
Under eager evaluation, we don't want our values to have anything in them that needs to take a step.



This is fine:

succ(pred(succ O))

because a natural number is either zero or the successor of a natural number.



with another succ on the outside.

succ (pred (succ *O*))

then the outer succ will be removed and evaluation will continue until we hit a O or end up

If we don't use pred on this:

then no one cared about it anyhow, so nothing of value is lost.

succ (pred (succ *O*))

```
data Term =
    TmZero
    | TmSucc Term
    | TmPred Term
    deriving (Eq, Ord, Show)
```

```
vZero :: Rule Term ()
vZero _ TmZero =
    Just ()
vZero _ _ =
    Nothing
(V-ZERO)
```

vSuccEager \_ \_ =
Nothing

```
valueEagerR :: RuleSet Term ()
valueEagerR =
  mkRuleSet [vZero, vSuccEager]
```

```
vSuccLazy :: Rule Term ()
vSuccLazy _ (TmSucc tm) =
  pure ()
vSuccLazy _ =
  Nothing
(V-Succ)
```

```
valueLazyR :: RuleSet Term ()
valueLazyR =
  mkRuleSet [vZero, vSuccLazy]
```

```
eSucc :: Rule Term Term
```

pure \$ TmSucc tm'

eSucc step (TmSucc tm) = do

 $rac{t_1 \longrightarrow t_1'}{\mathsf{succ}\ t_1 \longrightarrow \mathsf{succ}\ t_1'}$  (E-Succ)

tm' <- step tm

eSucc \_ \_ = Nothing

```
ePred :: Rule Term Term
ePred step (TmPred tm) = do
```

ePred \_ \_ = Nothing

 $rac{t_1 \longrightarrow t_1'}{\mathsf{pred} \ t_1 \longrightarrow \mathsf{pred} \ t_1'} \quad ext{(E-Pred)}$ 

tm' <- step tm pure \$ TmPred tm'

```
ePredZero :: Rule Term Term
ePredZero _ (TmPred TmZero) =
  Just TmZero
```

ePredZero \_ \_ =
 Nothing

 $\frac{}{\mathsf{pred}\ \mathsf{O}\longrightarrow\mathsf{O}}\,(\mathrm{E}\text{-}\mathrm{Pred}\mathrm{Zero})$ 

```
ePredSuccEager :: RuleSet Term () -> Rule Term Term
  _ <- value tm
```

 $\frac{}{\text{pred (succ } v) \longrightarrow v} \text{ (E-PREDSUCC)}$ 

```
ePredSuccEager value _ (TmPred (TmSucc tm)) = do
```

pure tm

Nothing

ePredSuccEager \_ \_ =

### stepEagerR :: RuleSet Term Term

stepEagerR =

mkRuleSet [eSucc, ePred, ePredZero, ePredSuccEager valueEagerR]

```
ePredSuccLazy :: Rule Term Term
ePredSuccLazy _ (TmPred (TmSucc tm)) =
```

Just tm

Nothing

ePredSuccLazy \_ \_ =

 $\frac{}{\text{pred (succ }t)\longrightarrow t}$  (E-PREDSUCC)

# stepLazyR :: RuleSet Term Term stepLazyR = mkRuleSet [ePred, ePredZero, ePredSuccLazy]

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
```

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepEagerR tm
```

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepEagerR tm
Just (TmSucc TmZero)
```

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepEagerR tm
```

Just (TmSucc TmZero)

> stepEagerR >=> stepEagerR \$ tm

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepEagerR tm
```

Just (TmSucc TmZero)

> stepEagerR >=> stepEagerR \$ tm

Nothing

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
```

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepLazyR tm
```

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepLazyR tm
```

Nothing

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
```

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
```

> stepEagerR \$ TmPred tm

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepEagerR $ TmPred tm
```

I. + (T. D. 1 (T. G. . T. T. 7 . .

Just (TmPred (TmSucc TmZero))

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
```

Just (TmPred (TmSucc TmZero))

> stepEagerR \$ TmPred tm

```
> stepEagerR >=> stepEagerR $ TmPred tm
```

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepEagerR $ TmPred tm
```

Just (TmPred (TmSucc TmZero))

Just TmZero

> stepEagerR >=> stepEagerR \$ TmPred tm

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepEagerR $ TmPred tm
```

> stepEagerR >=> stepEagerR \$ TmPred tm

Just (TmPred (TmSucc TmZero))

Just TmZero

> stepEagerR >=> stepEagerR >=> stepEagerR \$ TmPred tm

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepEagerR $ TmPred tm
Just (TmPred (TmSucc TmZero))
```

> stepEagerR >=> stepEagerR \$ TmPred tm

Just TmZero

> stepEagerR >=> stepEagerR >=> stepEagerR \$ TmPred tm

Nothing

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
```

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
```

> stepLazyR \$ TmPred tm

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepLazyR $ TmPred tm
```

- -

Just (TmPred (TmSucc TmZero))

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepLazyR $ TmPred tm
```

Just (TmPred (TmSucc TmZero))

> stepLazyR >=> stepLazyR \$ TmPred tm

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepLazyR $ TmPred tm
```

> stepLazyR >=> stepLazyR \$ TmPred tm

Just (TmPred (TmSucc TmZero))

Just TmZero

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepLazyR $ TmPred tm
```

Just (TmPred (TmSucc TmZero))

Just TmZero

> stepLazyR >=> stepLazyR \$ TmPred tm

> stepLazyR >=> stepLazyR >=> stepLazyR \$ TmPred tm

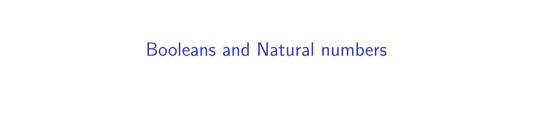
```
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepLazyR $ TmPred tm
Just (TmPred (TmSucc TmZero))
```

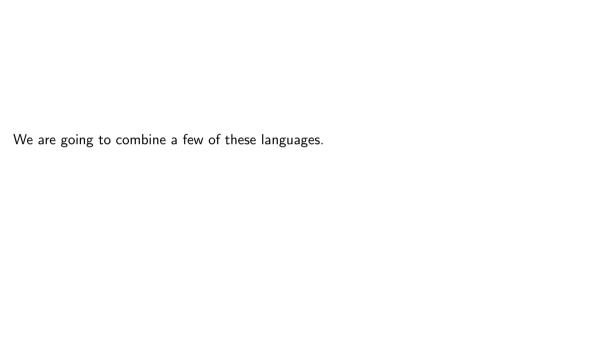
> stepLazyR >=> stepLazyR \$ TmPred tm

Just TmZero

> stepLazyR >=> stepLazyR >=> stepLazyR \$ TmPred tm

Nothing





#### **Terms**

```
false
                         constant false
true
                          constant true
                            disjunction
t or t
                          constant zero
succ t
                              successor
pred t
                            predecessor
iszero t
                                  iszero
if t then t else t
                                      if
```

## Small-step semantics for iszero (eager)

ro 
$$t \longrightarrow \mathsf{iszero}\ t'$$

$$\overline{\text{iszero O} \longrightarrow \text{true}}$$

iszero (succ v)  $\longrightarrow$  false

$$rac{t \longrightarrow t'}{\mathsf{iszero} \,\, t \longrightarrow \mathsf{iszero} \,\, t'}$$

(E-IsZeroSucc)

(E-IsZero)

## Small-step semantics for iszero (lazy)

$$rac{t \longrightarrow t'}{\mathsf{iszero} \ t \longrightarrow \mathsf{iszero} \ t'}$$

$$\overline{\mathsf{iszero} \ \mathsf{O} \longrightarrow \mathsf{true}}$$

iszero (succ t)  $\longrightarrow$  false

(E-IsZero)

(E-IsZeroSucc)

### Small-step semantics for if

 $t_1 \longrightarrow {t_1}'$ 

if  $t_1$  then  $t_2$  else  $t_3 \longrightarrow$  if  $t_1'$  then  $t_2$  else  $t_3$ 

if true then  $t_2$  else  $t_3 \longrightarrow t_2$ 

if false then  $t_2$  else  $t_3 \longrightarrow t_3$ 

(E-IfFalse)

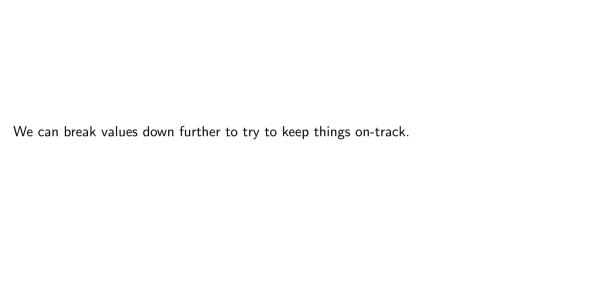
(E-IF)

This term is stuck:

iszero false

as is this one:

if O then false else true



## **Values**

nv

*bv* :=



false

true

0

bv

nv













succ nv







natural number value



false value true value



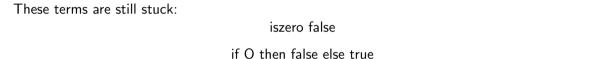


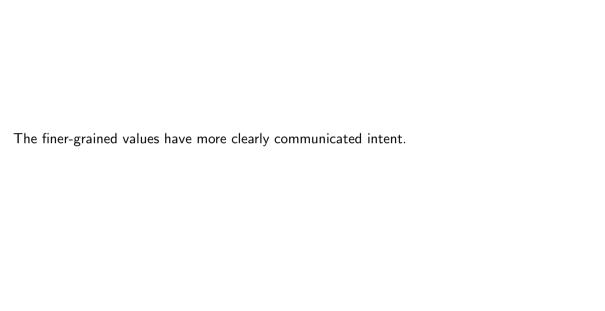


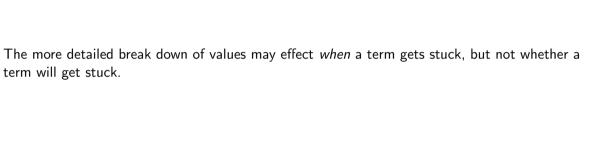
#### **Small-step semantics**

 $\overline{\mathsf{iszero} \; (\mathsf{succ} \; | \mathit{nv} \;) \longrightarrow \mathsf{false}}$ 

(E-IsZeroSucc)







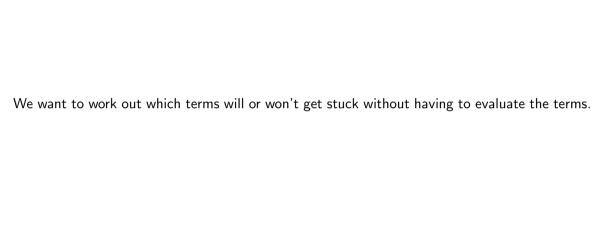
```
> check boolNatRulesLazyValueOrStep
  <interactive> failed after 14 tests and 5 shrinks.
```

```
src/Util/Rules.hs
    exactlyOne :: (Show a)
48
               => Gen a
49
               -> RuleSet a b
50
               -> RuleSet a c
51
               -> Property
52
   exactlyOne ga r1 r2 = property $ do
53
      a <- forAll ga
      TmOr TmZero TmZero
      isJust (r1 a) /== isJust (r2 a)
54
      | Both equal to
       False
```

#### False

```
src/Util/Rules.hs
    exactlyOne :: (Show a)
48
               => Gen a
49
               -> RuleSet a b
50
               -> RuleSet a c
51
               -> Property
52
   exactlyOne ga r1 r2 = property $ do
53
      a <- forAll ga
      | TmPred TmFalse
      isJust (r1 a) /== isJust (r2 a)
54
      | Both equal to
       False
```

#### False





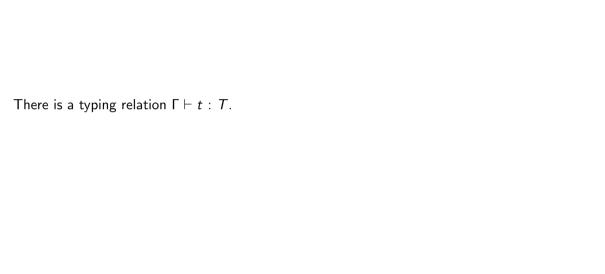
#### **Types**

:=

Bool Nat

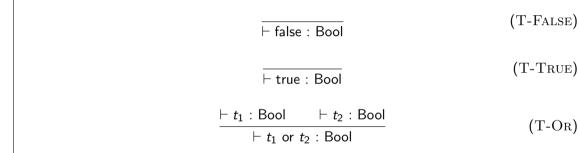
type of booleans

type of natural numbers



his indicates that the	e term $t$ is $$ well-typed	d and has type	${\cal T}$ inside the context $\epsilon$	of Γ.

At the moment we don't have any relevant context, so we work with $\vdash t:T$ .	



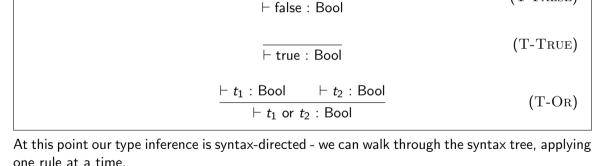
We can use these rules to check that a given term has a particular type.

Typing rules (Booleans)

⊢ false : Bool	(T-False)
⊢ true : Bool	(T-True)
$\frac{\vdash t_1 : Bool \qquad \vdash t_2 : Bool}{\vdash t_1 \text{ or } t_2 : Bool}$	(T-Or)

We can use these rules to *infer* the type for a particular term.

Typing rules (Booleans)



(T-False)

Typing rules (Booleans)

Typing rules (Natural numbers)	
$\overline{\vdash O : Nat}$	(T-Zero)
$\frac{\vdash t : Nat}{\vdash succ\ t : Nat}$	(T-Succ)
$\frac{\vdash t : Nat}{\vdash pred\ t : Nat}$	(T-Pred)

## Typing rules (both)

$$\frac{\vdash t : \mathsf{Nat}}{\vdash \mathsf{iszero}\ t : \mathsf{Bool}}$$

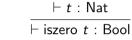
Bool

 $\frac{\vdash t_1 : \mathsf{Bool} \qquad \vdash t_2 : T \qquad \vdash t_3 : T}{\vdash \mathsf{if} \ t_1 \ \mathsf{then} \ t_2 \ \mathsf{else} \ t_3 : T}$ 

(T-IsZero)

(T-IF)

# Typing rules (both)



 $\vdash$  if  $t_1$  then  $t_2$  else  $t_3$ : T

iszero false

 $\vdash t_1 : \mathsf{Bool} \qquad \vdash t_2 : T \qquad \vdash t_3 : T$ 

(T-IsZero)

 $(T-I_F)$ 

These rule out the stuck terms we saw previously:

and

if O then false else true

Typing rules (both) 
$$\frac{\vdash t : \mathsf{Nat}}{\vdash \mathsf{iszero}\ t : \mathsf{Bool}} \tag{T-IsZero}$$
 
$$\frac{\vdash t_1 : \mathsf{Bool} \qquad \vdash t_2 : T \qquad \vdash t_3 : T}{\vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T} \tag{T-IF}$$

They also rule out terms that are not stuck:

if true then O else true

as the rule T-If states that both branches of the if have to have the same type.

$$\frac{\vdash t : \mathsf{Nat}}{\vdash \mathsf{iszero}\ t : \mathsf{Bool}}$$

 $\vdash t_1 : \mathsf{Bool} \qquad \vdash t_2 : T \qquad \vdash t_3 : T$ 

 $\vdash$  if  $t_1$  then  $t_2$  else  $t_3$ : T

(T-IsZero)

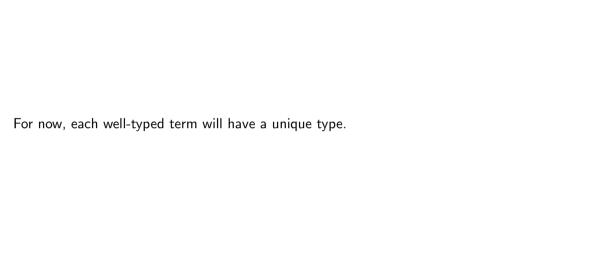
(T-IF)

$$\frac{|T-ISZERO|}{|T-ISZERO|}$$
 (T-ISZERO) 
$$\frac{|T-ISZERO|}{|T-ISZERO|}$$
 (T-IF) When a type system rules out some terms that were not stuck, it is called a *conservative* type

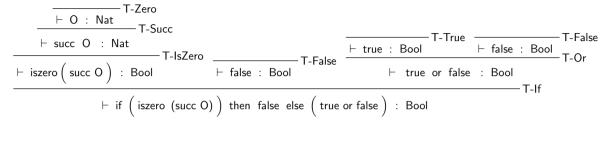
 $\vdash t$ : Nat

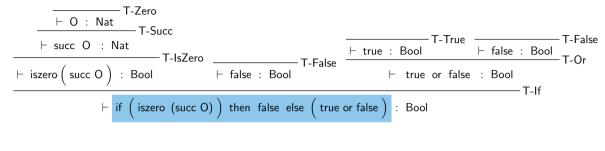
Typing rules (both)

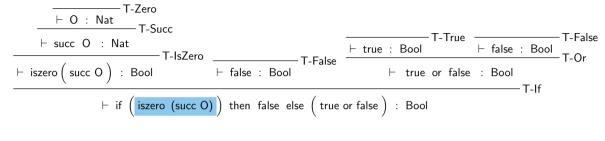
system.

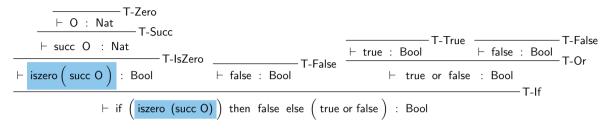


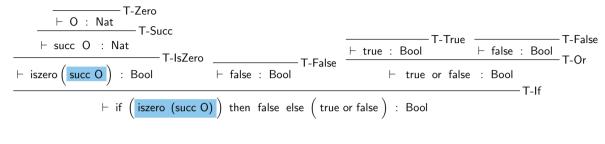
ater on that will relax, and we'll be more concerned with the <i>principal type</i> of a ter	m.

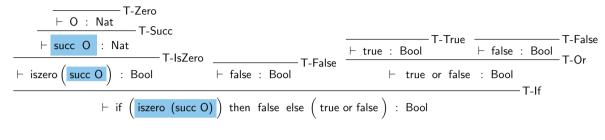


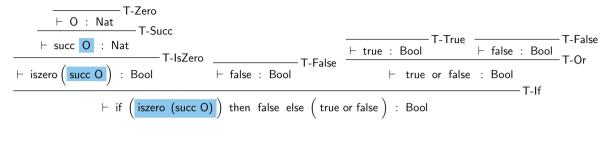


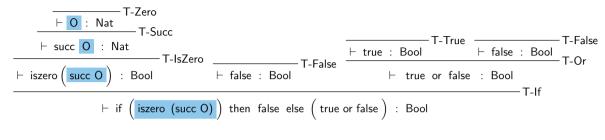


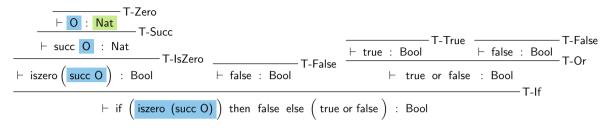


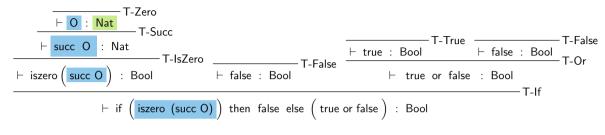


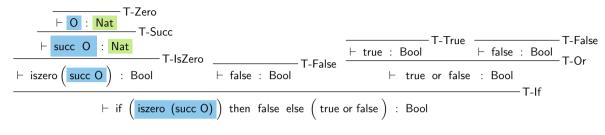


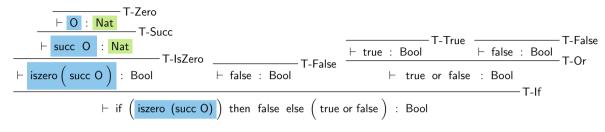


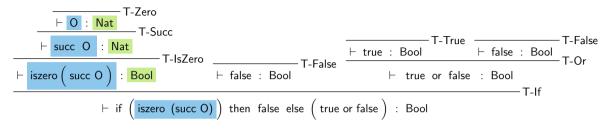


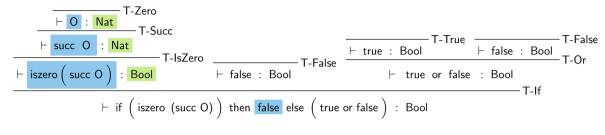


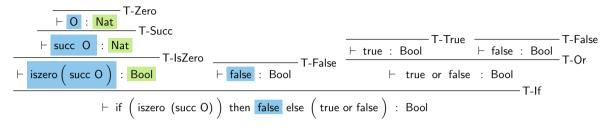


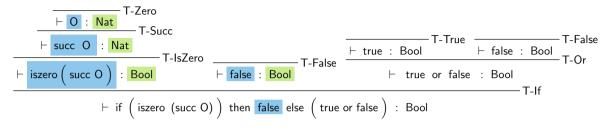


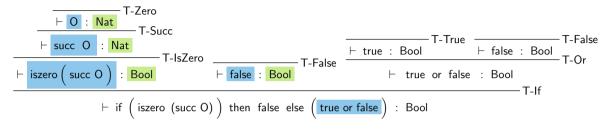


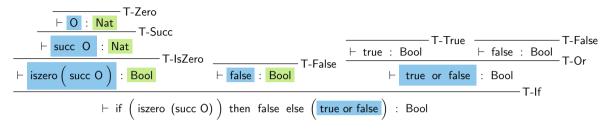


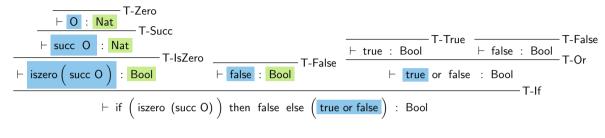


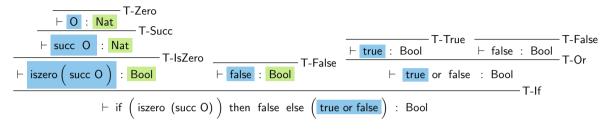


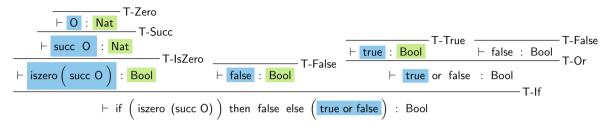


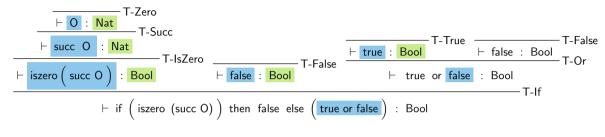


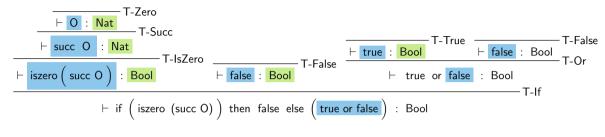


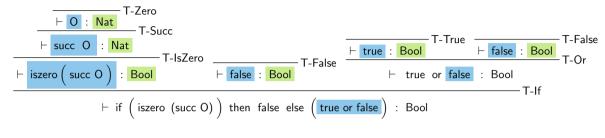


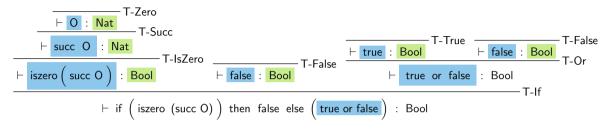


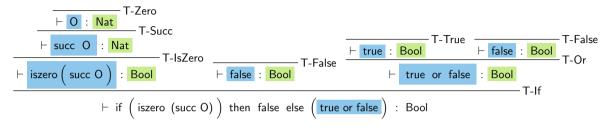


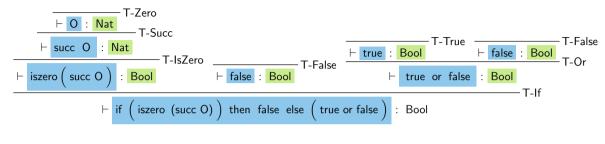


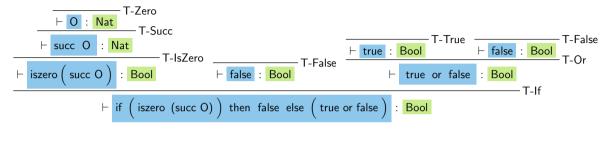












deriving (Eq, Ord, Show)

```
checkIsZero :: Rule (Term, Type) ()
checkIsZero step (TmIsZero tm, TyBool) =
  step (tm, TyNat)
checkIsZero _ _ =
```

Nothing

```
\frac{\vdash t : \mathsf{Nat}}{\vdash \mathsf{iszero} \ t : \mathsf{Bool}} \quad (\mathsf{T}\text{-}\mathsf{IsZero})
```

```
checkIf :: Rule (Term, Type) ()
checkIf step (TmIf tm1 tm2 tm3, ty) = do
  step (tm1, TyBool)
```

```
\frac{\vdash t_1 : \mathsf{Bool} \qquad \vdash t_2 : T \qquad \vdash t_3 : T}{\vdash \mathsf{if} \ t_1 \ \mathsf{then} \ t_2 \ \mathsf{else} \ t_3 : T} \, \mathsf{(T-IF)}
```

```
step (tm2, ty)
step (tm3, ty)
```

checkIf \_ \_ = Nothing

```
> let tmBad = TmIf (TmIsZero (TmSucc TmFalse)) TmFalse (TmOr TmTrue TmFalse)
```

```
> let tmBad = TmIf (TmIsZero (TmSucc TmFalse)) TmFalse (TmOr TmTrue TmFalse)
> checkType (tmBad, TyBool)
```

```
> let tmBad = TmIf (TmIsZero (TmSucc TmFalse)) TmFalse (TmOr TmTrue TmFalse)
> checkType (tmBad, TyBool)
Nothing
```

- > let tmBad = TmIf (TmIsZero (TmSucc TmFalse)) TmFalse (TmOr TmTrue TmFalse)
  > checkType (tmBad, TyBool)
- > let tmGood = TmIf (TmIsZero (TmSucc TmZero)) TmFalse (TmOr TmTrue TmFalse)

Nothing

- > let tmBad = TmIf (TmIsZero (TmSucc TmFalse)) TmFalse (TmOr TmTrue TmFalse)
  > checkType (tmBad, TyBool)
  Nothing
- > let tmGood = TmIf (TmIsZero (TmSucc TmZero)) TmFalse (TmOr TmTrue TmFalse)
- > checkType (tmGood, TyBool)

```
> let tmBad = TmIf (TmIsZero (TmSucc TmFalse)) TmFalse (TmOr TmTrue TmFalse)
> checkType (tmBad, TyBool)
Nothing
> let tmGood = TmIf (TmIsZero (TmSucc TmZero)) TmFalse (TmOr TmTrue TmFalse)
```

> checkType (tmGood, TyBool)

Just ()

```
> let tmBad = TmIf (TmIsZero (TmSucc TmFalse)) TmFalse (TmOr TmTrue TmFalse)
> checkType (tmBad, TyBool)
Nothing
> let tmGood = TmIf (TmIsZero (TmSucc TmZero)) TmFalse (TmOr TmTrue TmFalse)
```

> checkType (tmGood, TyBool)

> checkType (tmGood, TyNat)

Just ()

```
> let tmBad = TmIf (TmIsZero (TmSucc TmFalse)) TmFalse (TmOr TmTrue TmFalse)
> checkType (tmBad, TyBool)
Nothing
> let tmGood = TmIf (TmIsZero (TmSucc TmZero)) TmFalse (TmOr TmTrue TmFalse)
```

> checkType (tmGood, TyNat)

Just ()

Nothing

> checkType (tmGood, TyBool)

```
expect :: Eq ty => RuleSet tm ty -> tm -> ty -> Maybe ()
expect step tm ty = do
 ty' <- step tm
 if (ty == ty')
 then Just ()
 else Nothing
expectEq :: Eq ty => RuleSet tm ty -> tm -> tm -> Maybe ty
expectEq step tm1 tm2 = do
 ty1 <- step tm1
 ty2 <- step tm2
 if (ty1 == ty2)
 then Just ty1
```

else Nothing

```
inferIsZero :: Rule Term Type
inferIsZero step (TmIsZero tm) = do
  expect step tm TyNat
```

pure TyBool inferIsZero \_ =

Nothing

```
\frac{\vdash t : \mathsf{Nat}}{\vdash \mathsf{iszero} \ t : \mathsf{Bool}} \quad \text{(T-IsZero)}
```

```
inferIf :: Rule Term Type
```

inferIf step (TmIf tm1 tm2 tm3) = do

inferIf \_ \_ = Nothing

expect step tm1 TyBool

expectEq step tm2 tm3

 $\frac{\vdash t_1 : \mathsf{Bool} \qquad \vdash t_2 : T \qquad \vdash t_3 : T}{\vdash \mathsf{if} \ t_1 \ \mathsf{then} \ t_2 \ \mathsf{else} \ t_3 : T} \, (\mathsf{T}\text{-}\mathsf{IF})$ 

> :	let	tmBad	= TmI:	f (Tn	ıIsZero	(TmSucc	TmFalse))	TmFalse	(TmOr	TmTrue	TmFalse)

```
> let tmBad = TmIf (TmIsZero (TmSucc TmFalse)) TmFalse (TmOr TmTrue TmFalse)
> infer tmBad
```

```
> let tmBad = TmIf (TmIsZero (TmSucc TmFalse)) TmFalse (TmOr TmTrue TmFalse)
> infer tmBad
Nothing
```

```
> let tmBad = TmIf (TmIsZero (TmSucc TmFalse)) TmFalse (TmOr TmTrue TmFalse)
> infer tmBad
Nothing
```

> let tmGood = TmIf (TmIsZero (TmSucc TmZero)) TmFalse (TmOr TmTrue TmFalse)

```
> let tmBad = TmIf (TmIsZero (TmSucc TmFalse)) TmFalse (TmOr TmTrue TmFalse)
> infer tmBad
Nothing
```

> let tmGood = TmIf (TmIsZero (TmSucc TmZero)) TmFalse (TmOr TmTrue TmFalse)

> infer tmGood

```
> let tmBad = TmIf (TmIsZero (TmSucc TmFalse)) TmFalse (TmOr TmTrue TmFalse)
> infer tmBad
Nothing
> let tmGood = TmIf (TmIsZero (TmSucc TmZero)) TmFalse (TmOr TmTrue TmFalse)
```

> infer tmGood

Just TyBool

```
genType :: Gen Type
genType = ...
genTypedTerm :: Type -> Gen Term
genTypedTerm = ...
genWellTypedTerm :: Gen Term
```

genWellTypedTerm =

genType >>= genTypedTerm

```
checkTypeCorrect :: Property
checkTypeCorrect = property $ do
  (ty, tm) <- forAll $ do
   ty' <- genType
   tm' <- genTypedTerm ty'
   pure (ty', tm')
 checkType (tm, ty) === Just ()
inferCorrect :: Property
inferCorrect = property $ do
  (tv, tm) <- forAll $ do
   ty' <- genTvpe
   tm' <- genTypedTerm ty'
   pure (ty', tm')
  infer tm === Just tv
```

```
> check $ checkTypeCorrect
```

```
> check $ checkTypeCorrect
    <interactive> passed 100 tests.
True
```

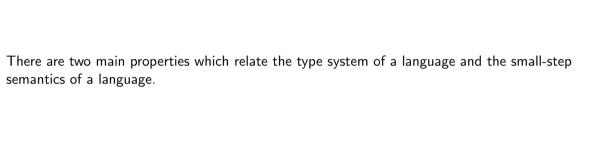
```
> check $ checkTypeCorrect
     <interactive> passed 100 tests.
True
```

> check \$ inferCorrect

```
> check $ checkTypeCorrect
    <interactive> passed 100 tests.
True
> check $ inferCorrect
```

<interactive> passed 100 tests.

True



 $\forall \vdash t:T \ \Big( \text{ value } t \lor \exists t'(t\longrightarrow t') \Big)$ 

 $\forall \vdash t:T \ ( \text{ value } t \lor \exists t'(t \longrightarrow t') \ )$ 

 $\forall \vdash t:T \ \left( \text{ value } t \lor \exists t'(t \longrightarrow t') \right)$ 

 $\forall \vdash t : T \ \Big( \text{ value } t \ \bigvee \ \exists t' \ (t \longrightarrow t') \ \Big)$ 

$$\forall \vdash t : T \ \left( \begin{array}{c|c} \mathsf{value} \ t \end{array} \lor \ \exists t' (t \longrightarrow t') \ \right)$$

 $\forall \vdash t:T \ \left( \text{ value } t \ \lor \ \exists t'(t \longrightarrow t') \ \right)$ 

$$\forall \vdash t:T \ \Big( \exists t' (t \longrightarrow t') \implies \vdash t':T \ \Big)$$

 $\forall \vdash t:T \ (\exists t'(t\longrightarrow t') \implies \vdash t':T)$ 

 $\forall \vdash t:T \ (\exists t'(t\longrightarrow t') \implies \vdash t':T)$ 

 $\forall \vdash t:T \ \left( \exists t' (t \longrightarrow t') \Longrightarrow \vdash t':T \right)$ 

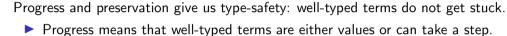
 $\forall \vdash t:T \mid \exists t'(t \longrightarrow t') \mid \implies \vdash t':T )$ 

 $\forall \vdash t:T \ \Big( \exists t' (t \longrightarrow t') \implies \boxed{\vdash t':T} \Big)$ 

Progress and preservation give us type-safety: well-typed terms do not get stuck.	



▶ Progress means that well-typed terms are either values or can take a step.



- ▶ Values are not stuck, so if we are at a value we are done.

Progress and preservation give us type-safety: well-typed terms do not get stuck.

- ▶ Progress means that well-typed terms are either values or can take a step.
- riogress means that wen typed terms are creater values of can take a step.

Values are not stuck, so if we are at a value we are done.

▶ Preservation means that well-typed terms that can take a step do not change type.

- Progress and preservation give us type-safety: well-typed terms do not get stuck.
- Progress means that well-typed terms are either values or can take a step.
- Values are not stuck, so if we are at a value we are done.
- Preservation means that well-typed terms that can take a step do not change type.
   After the step we have a well-typed term, so it is either a value or can take a step...

Progress and preservation also tie together syntax, semantics and typing.	

use type systems to (approxir	mately) o	classify v	vhich te	erms w	ill or

Together they mean we can

will not get stuck.

```
progress :: RuleSet Term () -> RuleSet Term Term -> Property
progress value step = property $ do
   tm <- forAll genWellTypedTerm
   isJust (value tm) /== isJust (step tm)

preservation :: RuleSet Term Term -> Property
preservation step = property $ do
   (tv. tm) <- forAll $ do</pre>
```

tm' <- Gen.filter (isJust . step) (genTypedTerm ty')</pre>

ty' <- genType

pure (tv', tm')

Just ty === (step tm >>= infer)

check \$ progress valu	ueEagerR stepEagerR	

>

> check \$ progress valueEagerR stepEagerR
 <interactive> passed 100 tests.
True

```
> check $ progress valueEagerR stepEagerR
<interactive> passed 100 tests.
```

### True

> check \$ preservation stepEagerR

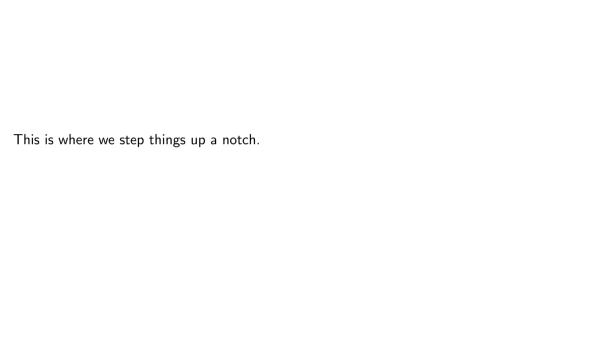
```
> check $ progress valueEagerR stepEagerR
     <interactive> passed 100 tests.
```

# True

> check \$ preservation stepEagerR
<interactive> passed 100 tests.

### True





#### Terms and values

Χ

t t

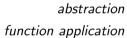
 $\lambda x.t$ 

 $\lambda x.t$ 















abstraction

variable

#### Lambda anatomy

 $\lambda x \cdot x + 2$ 

#### Lambda anatomy

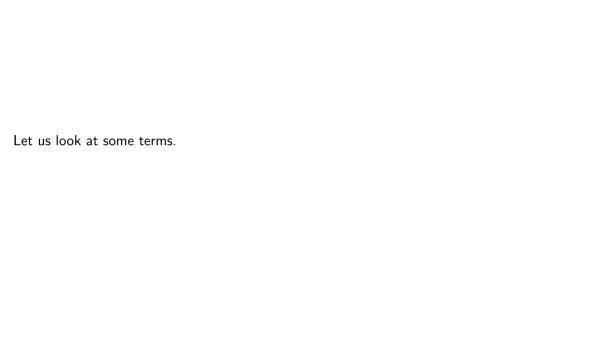
$$\lambda \times .x + 2$$

The x to the left of the . is called a variable binding.

#### Lambda anatomy

$$\lambda x \cdot x + 2$$

The  $\boldsymbol{x}$  to the right of the . is a variable.



The term

X

is meaningless.

The term

x + 2

is also meaningless.

The term

 $\lambda x \cdot x + 2$ 

f(x)=x+2

is an anonymous equivalent to

The term

when f is defined as

 $(\lambda x.x + 2) 1$ 

f(1)

f(x) = x + 2

is equivalent to

like before.

In ordinary maths, we evaluate

by taking

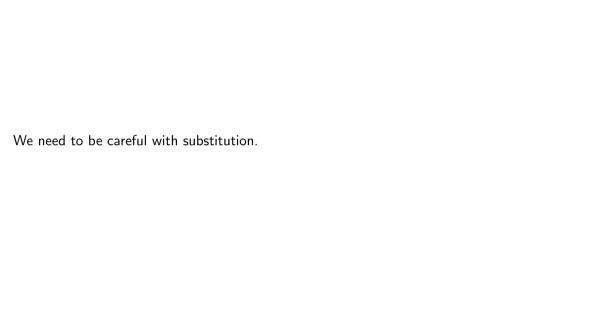
f(1) = 1 + 2

f(x) = x + 2and replacing the occurrences of x with 1 to get

ΓhΔ	notation	for	that	kind	$\circ$ f	renlacem	ent

 $[x \mapsto 1] f$ 

We would like to see something similar happening in our evaluation rules:
$(\lambda \times \times + 2)1 \longrightarrow 1 + 2$



When we evaluate

$$(\lambda x. (\lambda x.x + 1) (x + 1)) 3$$

we have



$$[x \mapsto 3] ((\lambda x.x + 1) (x + 1))$$

 $[x \mapsto 3]((\lambda x.x+1) (x+1))$ 

 $(\lambda x.x + 1) (3 + 1)$ 

 $(\lambda x.3 + 1) (3 + 1)$ 

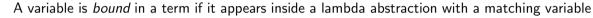
to become

instead of





 $((\lambda \times ... \times +1) (x+1))$ 



binding.

 $\lambda \times . \times + 1$ 

A variable can appear as both *free* and *bound* in the same term.

 $((\lambda \times . \times + 1) (x + 1))$ 

A variable can appear as both *free* and *bound* in the same term.

$$((\lambda \times . \times +1) (\times +1))$$

A variable can appear as both *free* and **bound** in the same term.

$$((\lambda \times . \times + 1) (x + 1))$$

In

$$[x \mapsto 3] ((\lambda x.x + 1) (x + 1))$$

we want to replace the occurrences of x as a free variable with 3.

We have the option of renaming the bound variables to fresh names

$$[x \mapsto 3]((\lambda z.z+1) (x+1))$$

[x + y = 0]((x + 1) + (x + 1))

since it doesn't change the meaning of the program.

# Free variable rules $FV(x) = \{x\}$ $FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$ $FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$

$$egin{array}{lll} \mathit{FV}(\langle \mathit{int} 
angle) & = \emptyset \ \mathit{FV}(t_1 + t_2) & = \mathit{FV}(t_1) \cup \mathit{FV}(t_2) \end{array}$$

 $FV((\lambda x.x + 1) (x + 1)) = ?$ 

$$FV(x) = \{x\}$$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$

$$FV(\langle int \rangle) = \emptyset$$

 $FV((\lambda x.x+1) (x+1)) = ?$ 

$$FV(\langle int \rangle) = \emptyset$$
  
 $FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$ 

$$/(t_2)$$

$$FV(x) = \{x\}$$

$$FV(\lambda \times t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$$

$$FV(\langle Int \rangle) = \emptyset$$
  
 $FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$ 

 $FV((\lambda x.x+1) (x+1)) = ?$ 

$$FV(\langle int \rangle) = \emptyset$$







$$/(t_2)$$



$$FV(x) = \{x\}$$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$(x,t_1) =$$

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$





 $FV((\lambda x.x+1) (x+1)) = ?$ 

 $FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$ 





 $FV(\langle int \rangle) = \emptyset$ 















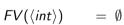
$$FV(x) = \{x\}$$

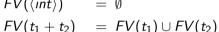
$$(x t_1) =$$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$(t_1) =$$

$$FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$$









$$FV(t_2)$$



#### Free variable rules $FV(x) = \{x\}$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1\ t_2) \qquad =\ FV(t_1) \cup FV(t_2)$$

$$FV(t_1+t_2) = FV(t_1) \cup FV(t_2)$$

$$FV(\langle int \rangle) = \emptyset$$

$$(t_2)$$

$$FV(x) = \{x\}$$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$FV(\langle int \rangle) = \emptyset$$
  
$$FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$$

$$FV(\langle int \rangle) = \emptyset$$



$$= FV(t_1) \cup$$

$$FV(t_1) \cup FV$$

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$





$$FV(x) = \{x\}$$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$

$$FV(\langle int \rangle) = \emptyset$$
  
$$FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$$

$$FV(\langle int \rangle) = \emptyset$$

$$\mathcal{FV}(t_2)$$



$$(t_2)$$

$$FV(x) = \{x\}$$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$(x,t_1) =$$

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$

$$FV(\langle Int \rangle) = \emptyset$$
  
$$FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$$

$$(t_2) =$$

$$FV(\langle int \rangle) = \emptyset$$

$$FV(t_2)$$

$$FV(t_2)$$

$$FV(t_2)$$

### Free variable rules $FV(x) = \{x\}$ $FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$

$$FV(\langle int \rangle) = \emptyset$$

$$FV(\langle int \rangle) = \emptyset$$
  
$$FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$$

$$t_2$$
) =

$$= \emptyset$$
  
 $= FV(t_1)$ 

$$= \emptyset$$
$$= FV(t_1) \bot$$

 $FV((\lambda x.x+1) (x+1)) = FV(\lambda x.x+1) \cup FV(x+1)$ 

 $= (FV(x+1) \setminus \{x\}) \cup FV(x+1)$ 

```
Free variable rules
FV(x) = \{x\}
FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}
FV(t_1 t_2) = FV(t_1) \cup FV(t_2)
FV(\langle int \rangle) = \emptyset
FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)
```

 $= (FV(x+1) \setminus \{x\}) \cup FV(x+1)$ 

#### Free variable rules $FV(x) = \{x\}$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$

$$FV(\langle int \rangle) = \emptyset$$

$$FV(\langle int \rangle) = \emptyset$$
  
$$FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$$

 $FV((\lambda x.x+1) (x+1)) = (FV(x+1) \setminus \{x\}) \cup FV(x+1)$ 

$$FV(t_2)$$

$$FV(t_2)$$

$$V(t_2)$$

# Free variable rules $FV(x) = \{x\}$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$

$$t_2$$
) =

$$FV(t_1+t_2) = FV(t_1) \cup FV(t_2)$$

$$FV(\langle int \rangle) = \emptyset$$

 $FV((\lambda x.x+1) (x+1)) = (FV(x+1) \setminus \{x\}) \cup FV(x+1)$ 

= FV(x+1)

$$FV(t_2)$$

### Free variable rules $FV(x) = \{x\}$

$$FV(\lambda \times t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$

$$=$$

$$FV(\langle int \rangle) = \emptyset$$
  
 $FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$ 

$$\rangle) = t_0 = t_0$$

$$FV(\langle int \rangle) = \emptyset$$

$$FV(x) = \{x\}$$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

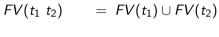
$$= FV(t_1$$

$$\rangle)$$
 =

$$FV(\langle Int \rangle) = \emptyset$$
  
 $FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$ 

$$FV(\langle int \rangle) = \emptyset$$







$$FV(t_1) \cup FV(t_1)$$

$$=V(t_2)$$

$$FV(x) = \{x\}$$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$



$$|t\rangle)$$

$$FV(\langle int \rangle) = \emptyset$$
  
 $FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$ 

$$FV(\langle int \rangle) = \emptyset$$

$$FV(x) = \{x\}$$

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix}$$
  $F$ 

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$

$$FV(\langle int \rangle) = \emptyset$$
  
 $FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$ 

$$FV(\langle int \rangle) = \emptyset$$

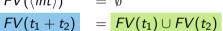
$$V(t_2)$$

$$FV(x) = \{x\}$$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$

$$FV(\langle int \rangle) = \emptyset$$







 $FV((\lambda x.x+1) (x+1)) = FV(x+1)$ 

 $= FV(x) \cup FV(1)$ 

```
Free variable rules
 FV(x) = \{x\}
 FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}
```

$$= \{ j \in \mathbb{Z} \}$$

$$= FV(t)$$

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$

 $FV((\lambda x.x + 1) (x + 1)) = FV(x + 1)$ 

$$t_2$$
) =

$$-t_2) =$$

$$FV(\langle int \rangle) = \emptyset$$

$$FV(\langle int \rangle) = \emptyset$$
  
 $FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$ 

 $= FV(x) \cup FV(1)$ 

# Free variable rules $FV(x) = \{x\}$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$
  
$$FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$$

$$FV(\langle int \rangle) = \emptyset$$
  
 $FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$ 

$$FV(\langle int \rangle) = \emptyset$$

$$= \emptyset$$

$$FV(x) = \{x\}$$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$= FV(t_1)$$

$$FV(\langle int \rangle) = \emptyset$$
  
 $FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$ 

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$

$$= F$$

$$V(t_2)$$

$$FV(x) = \{x\}$$

$$= FV($$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$

$$\langle int 
angle$$
)

$$\langle int \rangle$$
)

$$FV(t_1+t_2) = FV(t_1) \cup FV(t_2)$$

$$t_1 + t_2$$

$$FV(\langle int \rangle) = \emptyset$$











$$\{Y(t_0)\}$$

$$FV(x) = \begin{cases} x \\ FV(\lambda x.t_1) \end{cases} = FV(t_1) \setminus \{x\}$$

$$(x,t_1) = FV$$

$$FV(t_1 \ t_2)$$
 =  $FV(t_1) \cup FV(t_2)$ 

$$= r v$$

$$FV(\langle int \rangle) = \emptyset$$

$$FV(\langle int \rangle) = \emptyset$$
  
 $FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$ 

### Free variable rules $FV(x) = \{x\}$ $FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$

$$V(t_1 V(t_1$$

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$

$$nt\rangle$$
) =

$$) = \emptyset$$

$$FV(\langle int \rangle) = \emptyset$$

$$FV(\langle int \rangle) = \emptyset$$
  
 $FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$ 

 $= \{x\} \cup FV(1)$ 

$$FV((\lambda x.x+1) (x+1)) = FV(x) \cup FV(1)$$

```
Free variable rules

FV(x) = \{x\}
FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}
FV(t_1 t_2) = FV(t_1) \cup FV(t_2)
FV(\langle int \rangle) = \emptyset
FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)
```

$$FV((\lambda x.x + 1) (x + 1)) = FV(x) \cup FV(1)$$
  
=  $\{x\} \cup FV(1)$ 

### Free variable rules $FV(x) = \{x\}$ $FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$ $FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$

$$V(t_1 \ t_2) = FV(t_1)$$

$$FV(\langle int \rangle) = \emptyset$$

$$FV(\langle int \rangle) = \emptyset$$
  
 $FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$ 

$$FV((\lambda x.x+1) (x+1)) = \{x\} \cup FV(1)$$

### Free variable rules $FV(x) = \{x\}$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$

 $FV((\lambda x.x + 1) (x + 1)) = \{x\} \cup FV(1)$ 

$$FV(t_1+t_2) = FV(t_1) \cup FV(t_2)$$

$$FV(\langle int \rangle) = \emptyset$$

$$FV(x) = \{x\}$$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$

$$\begin{array}{ll}
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{array}$$

$$FV(\langle int \rangle) = \emptyset$$

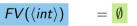
 $FV((\lambda x.x + 1) (x + 1)) = \{x\} \cup FV(1)$ 

$$-V(t_2)$$

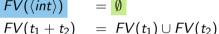
### Free variable rules $FV(x) = \{x\}$

$$FV(x) = \{x\}$$
  
$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$$







$$=\emptyset$$

 $FV((\lambda x.x + 1) (x + 1)) = \{x\} \cup FV(1)$ 

```
Free variable rules

FV(x) = \{x\}
FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}
FV(t_1 t_2) = FV(t_1) \cup FV(t_2)
```

$$FV((\lambda x.x+1) (x+1)) = \{x\} \cup FV(1)$$
$$= \{x\} \cup \emptyset$$

 $FV(t_1+t_2) = FV(t_1) \cup FV(t_2)$ 

 $FV(\langle int \rangle) = \emptyset$ 

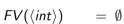
$$FV(x) = \{x\}$$

$$FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$$

$$=$$

$$FV(\langle Int \rangle) = \emptyset$$
  
 $FV(t_1 + t_2) = FV(t_1) \cup FV(t_2)$ 

$$+t_2$$

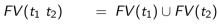






 $FV((\lambda x.x + 1) (x + 1)) = \{x\} \cup \emptyset$ 













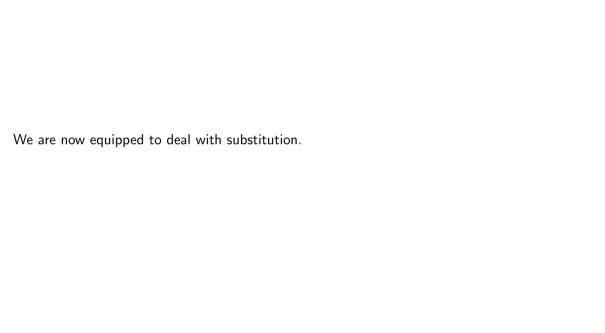
## Free variable rules $FV(x) = \{x\}$ $FV(\lambda x.t_1) = FV(t_1) \setminus \{x\}$ $FV(t_1 \ t_2) = FV(t_1) \cup FV(t_2)$

$$FV(\langle int \rangle) = \emptyset$$

 $FV((\lambda x.x + 1) (x + 1)) = \{x\}$ 

$$FV(\langle int \rangle) = \emptyset$$

$$FV(t_1 + t_2) = FV(t_3 + t_4) = FV(t_4 + t_5)$$



#### Substitution rules $[x \mapsto s] x$ = s $[x \mapsto s] \ v = v$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

 $[x \mapsto 3] ((\lambda x.x + 1) (x + 1)) = ?$ 

$$= y$$
 $= \lambda x$ 

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$\lambda y.([x])$$

$$([x \mapsto s]$$

$$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

$$([x \mapsto$$

$$=$$
 ([ $x \mapsto$ 

$$([x \mapsto s]$$

$$\epsilon\mapsto s]\ t_1)$$

$$t_1$$
) ([x

if 
$$y \neq x \land y \notin FV(s)$$

if 
$$y \neq x$$

#### Substitution rules $[x \mapsto s] x$ = s $[x \mapsto s] \ y$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

 $[x \mapsto 3] ((\lambda x.x + 1) (x + 1)) = ?$ 

$$= y$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$
  
 $[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] (\lambda y.t_1)$ 

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$
$$= \lambda y.([x \mapsto s] t_1)$$

$$y.([x \mapsto s]$$
  
 $[x \mapsto s] t_1)$ 

$$[x \mapsto s] (\lambda y. t_1) = \lambda y. ([x \mapsto s] t_1)$$
$$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

$$\rightarrow s] t_1) ([$$

$$[\mapsto s] \ t_1) \ ([x \mapsto s] \ t_1) \ ([x \mapsto s] \ t_2) \ ([x \mapsto s] \ t_3) \ ([x \mapsto s] \ t_3))$$

$$if y \neq x \land y \notin FV(s)$$

$$\mapsto s! t_2)$$

if 
$$y \neq x$$

#### Substitution rules $[x \mapsto s] x$ = s $[x \mapsto s] \ y$ = v

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

 $[x \mapsto 3] ((\lambda x.x + 1) (x + 1)) = ?$ 

$$[x \mapsto s] y = y$$
  
 $[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$ 

. 
$$t_1$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y. ([x \mapsto s] t_1)$$

$$\lambda y. ([x])$$

$$= \lambda y. ([x \vdash s])$$

$$([x\mapsto s]$$

$$[x \mapsto s] \ \ (t_1 \ t_2) \qquad = \ ([x \mapsto s] \ t_1) \ \ ([x \mapsto s] \ t_2)$$

$$([x\mapsto s]\ t_1)$$
 ([

$$([x \mapsto$$

if 
$$y \neq x \land$$

if 
$$y \neq x \land y \notin FV(s)$$

if 
$$y \neq x$$

#### Substitution rules $[x \mapsto s] x$ = s $[x \mapsto s] y$ = v $[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$

$$= \lambda x.t$$

$$\rightarrow s | t_1$$

$$[x \mapsto s]$$

$$\lambda y. ([x])$$

$$t_1)$$

$$s]t_2$$

$$[x \mapsto s] \quad (t_1 \quad t_2) \qquad = \quad ([x \mapsto s] \ t_1) \quad ([x \mapsto s] \ t_2)$$

 $[x \mapsto 3] ((\lambda x.x + 1) (x + 1)) = ?$ 

$$[x \mapsto s] \langle int \rangle = \langle int \rangle$$

$$([x\mapsto s]t_1)$$
  $([$ 

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$
$$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

$$s]\ t_1)$$

if 
$$y \neq x \land y \notin FV(s)$$

if 
$$y \neq x$$

$$[x \mapsto s] x$$

$$[x \mapsto s] x = s [x \mapsto s] y = y$$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

$$= y$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

 $[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$ 

$$= \lambda y.([x$$

$$\lambda y.([x$$

$$([ \lor \vdash \lor e] \cdot$$

$$[x \mapsto s] t_1)$$

$$\rightarrow s \mid t_1)$$
 (

$$\Rightarrow s \mid t_1$$
) ([x

$$[x \mapsto s] \ (t_1 \ t_2) = ([x \mapsto s] \ t_1) \ ([x \mapsto s] \ t_2)$$

 $[x \mapsto 3]$   $((\lambda x.x + 1) (x + 1))$  =  $([x \mapsto 3] (\lambda x.x + 1)) ([x \mapsto 3] (x + 1))$ 

$$x\mapsto s \mid t_2)$$

if 
$$y \neq x \land y \notin FV(s)$$

$$[x \mapsto s] x$$

$$[x \mapsto s] \ x = s [x \mapsto s] \ y = y$$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

$$= y$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$\lambda y.([x$$

$$= ([x \mapsto$$

 $[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$ 

$$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

$$([x\mapsto s]$$

$$\mapsto s] t_1)$$

$$\mapsto$$
 s]  $t_1$ ) (|

 $[x \mapsto 3] \ ((\lambda \ x.x+1) \ (x+1)) = ([x \mapsto 3] (\lambda \ x.x+1)) \ ([x \mapsto 3] (x+1))$ 

$$[t_2)$$

$$f y \neq x \wedge y$$

if 
$$y \neq x \land y \notin FV(s)$$

$$[x \mapsto s] x$$

$$[x \mapsto s] \ x = s$$

$$[x \mapsto s] \ y = y$$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

$$= y$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

 $[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$ 

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$\lambda y.([x$$

$$\lambda y.([x$$

$$\lambda y.([x$$

$$([x + 7], 0]$$

$$[x \mapsto s] (t_1, t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

 $[x \mapsto 3] ((\lambda x.x + 1) (x + 1)) = ([x \mapsto 3] (\lambda x.x + 1)) ([x \mapsto 3] (x + 1))$ 

$$[x \mapsto s] t_2$$

if 
$$y \neq x \land y \notin FV(s)$$

 $[x \mapsto s] \ y$ 

$$[x \mapsto s] x = s$$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

$$= y$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

 $[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$ 

$$\lambda y.([x$$

$$\langle y \cdot ([x \mid x) ) \rangle$$

$$f: ([x \mapsto s] t_1)$$

$$([ \lor \sqcup ]$$

$$\lambda y. ([x \vdash e]$$

$$y \cdot ([x \mapsto s] + 1)$$

$$\Rightarrow s] t_1)$$
 ([5

$$[x \mapsto s] (\lambda y. t_1) = \lambda y. ([x \mapsto s] t_1)$$
$$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

 $[x \mapsto 3] ((\lambda x.x + 1) (x + 1)) = ([x \mapsto 3] (\lambda x.x + 1)) ([x \mapsto 3] (x + 1))$ 

$$([x \mapsto s] t_2)$$

if 
$$y \neq x \land y \notin FV(s)$$

$$[x \mapsto s] x$$

$$[x \mapsto s] \ x = s$$

$$[x \mapsto s] \ y = y$$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

$$y = y$$

$$[x \mapsto s] (\lambda x.t_1) = \frac{\lambda x.t_1}{}$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$= \lambda y.([x$$

$$= \lambda y.([x$$

$$([x \mapsto s] t_1$$

$$([x\mapsto s]\ t_1)$$

$$[x\mapsto s]$$
  $(t_1 \ t_2)$   $=$   $([x\mapsto s] \ t_1)$   $([x\mapsto s] \ t_2)$ 

$$\rightarrow$$
 sj  $t_1$ ) ([ $z$ 

$$[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$$

 $[x \mapsto 3] ((\lambda x.x + 1) (x + 1)) = ([x \mapsto 3] (\lambda x.x + 1)) ([x \mapsto 3] (x + 1))$ 

$$\mapsto s$$
] ( $t_1$   $t_2$ 

$$\lambda x.t_1$$

if 
$$y \neq x$$

if 
$$y \neq x \land y \notin FV(s)$$

### Substitution rules $[x \mapsto s] x$ $[x \mapsto s] y$

$$\begin{array}{ccc} s & = s \\ y & = y \end{array}$$

$$[x \mapsto s] y = y$$
  
 $[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$ 

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$
$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$\lambda y.([x])$$

$$[x \mapsto s] (t_1 \ t_2) \qquad = ([x \mapsto s] \ t_1) ([x \mapsto s] \ t_2)$$

$$= ([x \mapsto$$

$$[x \mapsto s] \langle int \rangle = \langle int \rangle$$

$$\langle mt \rangle = \langle mt \rangle$$
  
 $(t_1 + t_2) = ([x \mapsto s])$ 

$$(t_1+t_2) = ([x\mapsto s])$$

$$[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$$

$$+([x\mapsto s]$$

 $[x \mapsto 3] ((\lambda x.x + 1) (x + 1)) = ([x \mapsto 3] (\lambda x.x + 1)) ([x \mapsto 3] (x + 1))$ 

$$\rightarrow s] t_2)$$

 $= \left( \lambda x.x + 1 \right) \left( \left[ x \mapsto 3 \right] \left( x + 1 \right) \right)$ 

if 
$$y \neq x \land y \notin FV(s)$$

### Substitution rules $[x \mapsto s] x$

$$[x \mapsto s] \ x = s [x \mapsto s] \ y = y$$

$$= y$$
 $- \lambda y$ 

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$= ([x \mapsto s]$$

$$= ([x \mapsto$$

 $[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$ 

$$[x \mapsto s] \langle int \rangle = \langle int \rangle$$

$$([x\mapsto s]\ t_1)$$

$$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

 $[x \mapsto 3] \ ((\lambda \ x.x+1) \ (x+1)) = ([x \mapsto 3] (\lambda \ x.x+1)) \ ([x \mapsto 3] (x+1))$ 

$$\mapsto s] t_2)$$

 $= (\lambda x.x + 1) ([x \mapsto 3] (x + 1))$ 

if 
$$y \neq x \land y \notin FV(s)$$

 $[x \mapsto s] v$ 

$$[x \mapsto s] x = s$$

$$= y$$

$$= y$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.(1)$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

$$t_1 t_2) = ($$

$$(i_1 i_2) - (i_2 i_3)$$

 $[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$ 

$$\int ([x \mapsto s] t$$

 $[x \mapsto 3] ((\lambda x.x+1) (x+1)) = (\lambda x.x+1) ([x \mapsto 3] (x+1))$ 

$$\kappa\mapsto s]t_2)$$

$$[t_2]$$

if 
$$y \neq x \land y \notin FV(s)$$

if 
$$y \neq x$$

$$[x \mapsto s] x = s$$

$$[x \mapsto s] \ y =$$

$$= y$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$[x \mapsto s] (t_1 \ t_2) \qquad = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

$$(t_1 t_2) = ([x +$$

$$(\iota_1 \ \iota_2) = ([x \vdash$$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

$$= ([x \mapsto$$

 $[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$ 

 $[x \mapsto 3] ((\lambda x.x + 1) (x + 1)) = (\lambda x.x + 1) ([x \mapsto 3] (x + 1))$ 

$$([x \mapsto s] t_2)$$

$$(x \mapsto s] t_2)$$

if 
$$y \neq x \land y \notin FV(s)$$

if 
$$y \neq y$$

if 
$$y \neq x$$

 $[x \mapsto s] \ y$ 

$$[x \mapsto s] x = s$$

$$= v$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s]t_1)$$

$$[x \mapsto s] (t_1, t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

$$= ([x \mapsto ]$$

$$([x\mapsto s]$$

$$([x\mapsto s]\ t_1$$

$$([x \mapsto s] t_1)$$

$$([x \mapsto s] t_1)$$

$$([x \mapsto s] t_2)$$

if  $y \neq x$ 

if  $y \neq x \land y \notin FV(s)$ 

$$[s] t_1) + ([x \mapsto s] t_2)$$

$$[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$$

$$(\lambda x.x+1) (x+1) = (\lambda x.x+1) ([x \vdash$$

$$[x \mapsto 3] ((\lambda x.x + 1) (x + 1)) = (\lambda x.x + 1) ([x \mapsto 3] (x + 1))$$

 $[x \mapsto s] \ y$ 

$$[x \mapsto s] x = s$$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

$$- s$$
 $= y$ 

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$\lambda y$$
. ([ $x$ 

$$= \lambda y \cdot ([x]$$

 $[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$ 

$$= \lambda y \cdot ([x]$$

$$\rightarrow s \mid t_1$$
) (

$$s] t_1) ([$$

 $[x \mapsto 3] ((\lambda x.x + 1) (x + 1)) = (\lambda x.x + 1) ([x \mapsto 3] (x + 1))$ 

$$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

$$\mapsto slt_2$$
)

if 
$$y \neq x \land y \notin FV(s)$$

### Substitution rules $[x \mapsto s] x$

$$[x \mapsto s] \ x = s [x \mapsto s] \ y = y$$

$$= y$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$\lambda y.([x \vdash$$

$$y \cdot ([x \mapsto s] t$$

$$[x \mapsto s] \ (t_1 \ t_2) \qquad = \ ([x \mapsto s] \ t_1) \ \ ([x \mapsto s] \ t_2)$$

$$= ([x \mapsto$$

$$=$$
  $\langle int \rangle$ 

$$+ t_2$$
 =  $([x \mapsto$ 

$$[x \mapsto s] \langle int \rangle = \langle int \rangle$$

$$\langle int \rangle$$

$$s]t_2)$$

if  $y \neq x$ 

if  $y \neq x \land y \notin FV(s)$ 

$$[x \mapsto 3] \; ((\lambda \; x.x+1) \; \; (x+1)) \quad = \; \Big( \; \lambda \; x.x+1 \; \Big) \; \; \Big([x \mapsto 3] \, \Big( x+1 \; \Big)\Big)$$

$$= (\lambda x.x + 1) ([x \mapsto 3] (x + 1))$$
$$= (\lambda x.x + 1) ([x \mapsto 3] x + [x \mapsto 3] 1)$$

$$(t_1+t_2) = ([$$

$$[x \mapsto s] \langle int \rangle = \langle int \rangle$$

$$[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$$

## Substitution rules $[x \mapsto s] x$

 $[x \mapsto s] y$ 

$$= s$$
  
 $= y$ 

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$
$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$\lambda y.([x])$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$
$$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

$$([x \mapsto s])$$

$$\langle int 
angle$$

$$[x \mapsto s] \ (t_1 + t_2) = ([x \mapsto s] \ t_1) + ([x \mapsto s] \ t_2)$$

 $[x \mapsto 3] \ ((\lambda \ x.x+1) \ (x+1)) = (\lambda \ x.x+1) \ ([x \mapsto 3] \ (x+1))$ 

$$[x \mapsto s] \langle int \rangle = \langle int \rangle$$

 $= (\lambda x.x + 1) ([x \mapsto 3] x + [x \mapsto 3] 1)$ 

if 
$$y \neq x \land y \notin FV(s)$$

 $[x \mapsto s] \ v$ 

$$[x \mapsto s] x = s$$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

$$= v$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$\lambda y.([x+]$$

$$([x \mapsto s])$$

 $[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$ 

$$[x \mapsto s] (t_1 \ t_2) \qquad = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

$$([x \mapsto s])$$

$$x\mapsto s]t_1$$

$$]t_1)$$
 ([x  $\vdash$ 

 $[x \mapsto 3]$   $((\lambda x.x+1)$  (x+1)) =  $(\lambda x.x+1)$   $([x \mapsto 3]$  x +  $[x \mapsto 3]$  1)

if 
$$y \neq x \land y \notin FV(s)$$
  
([ $x \mapsto s$ ]  $t_2$ )

$$\mapsto s]t_2)$$

$$|t_2|$$

 $[x \mapsto s] \ v$ 

$$[x \mapsto s] x = s$$

$$= y$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$= \lambda y.([x$$

 $[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$ 

$$([x \mapsto s]$$

$$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

 $[x \mapsto 3]$   $((\lambda x.x+1)$  (x+1)) =  $(\lambda x.x+1)$   $([x \mapsto 3]$   $x + [x \mapsto 3]$  1)

$$[x \mapsto s] \langle int \rangle = \langle int \rangle$$

$$\rightarrow$$
 S]  $\iota_1$ )

$$s\mapsto s]t_2$$

if 
$$y \neq x \land y \notin FV(s)$$

$$y \neq x \land y \notin$$

$$[x \mapsto s] x = s$$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

$$[x \mapsto s] \ y \qquad = y$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$= \lambda y.([x$$

 $[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$ 

$$([x \mapsto$$

$$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

$$= ([x \mapsto s])$$

$$([x \mapsto x])$$

$$x\mapsto s]t_1)$$

 $[x \mapsto 3]$   $((\lambda x.x + 1)$   $(x + 1)) = (\lambda x.x + 1)$   $([x \mapsto 3] \times + [x \mapsto 3] \times 1)$ 

$$\mapsto s_1 \iota_2 j$$

$$t_2$$
)

If 
$$y \neq z$$

if 
$$y \neq x \land y \notin FV(s)$$

$$[x \mapsto s] x = s [x \mapsto s] y = y$$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

$$= y$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$\lambda v.([$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$\langle x \rangle \cdot (x)$$

$$([x \mapsto$$

$$([x \mapsto$$

 $[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$ 

$$([x \mapsto s])$$

$$([x\mapsto s]$$

$$([x \mapsto s]$$

$$\rightarrow s$$
]  $t_1$ ) ([x

$$[x \mapsto s] (t_1 \ t_2) \qquad = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

 $[x \mapsto 3]$   $((\lambda x.x+1)$  (x+1)) =  $(\lambda x.x+1)$   $([x \mapsto 3]$   $x + [x \mapsto 3]$  1)

$$([x \mapsto s] t_2)$$

if 
$$y \neq x \land y \notin FV(s)$$

if 
$$y \neq x$$

## Substitution rules $[x \mapsto s] x = s$ $[x \mapsto s] \ v = v$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_2) = \lambda y.(f)$$

$$\lambda y. ([x$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y. ([x \mapsto s] t_1)$$
$$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_1)$$

$$([x\mapsto s]$$

$$= ([x \mapsto s])$$

$$\langle \mathit{int} 
angle$$

$$[x \mapsto s] (mt) = (mt)$$
$$[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$$

 $[x \mapsto 3]$   $((\lambda x.x + 1)$  (x + 1)) =  $(\lambda x.x + 1)$   $([x \mapsto 3]$   $x + [x \mapsto 3]$  1)

$$[x \mapsto s] \langle int \rangle = \langle int \rangle$$

$$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

$$s\mapsto s]\ t_1)$$

 $= (\lambda x.x + 1) (3 + [x \mapsto 3] 1)$ 

if 
$$y \neq x$$

if 
$$y \neq x \land y \notin FV(s)$$

#### Substitution rules $[x \mapsto s] x$ = s $[x \mapsto s] y$

$$= y$$

$$= y$$
  
 $= \lambda x.$ 

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$
$$[x \mapsto s] (\lambda y.t_1) = \lambda y.(1)$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y. ([x \mapsto s] t_1)$$
$$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

$$[x \mapsto s] \langle int \rangle = \langle int \rangle$$

$$|nt\rangle = |$$

$$[x \mapsto s] \langle int \rangle = \langle int \rangle [x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$$

$$=\langle \mathit{int} \rangle$$

 $[x \mapsto 3]$   $((\lambda x.x+1)$  (x+1)) =  $(\lambda x.x+1)$   $([x \mapsto 3]$   $x + [x \mapsto 3]$  1)

$$s t_2$$

 $= \left(\lambda x.x+1\right) \left(3 + [x \mapsto 3] 1\right)$ 

$$\land y \notin FV$$

if 
$$y \neq x \land y \notin FV(s)$$

 $[x \mapsto s] v$ 

$$[x \mapsto s] x = s$$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

$$= y$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$\lambda y.([x$$

$$= \lambda y \cdot ([x]$$

$$= ([x \mapsto ]$$

 $[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$ 

$$([x\mapsto s]$$

$$x\mapsto s]\ t_1)$$

$$\rightarrow s] t_1) ([$$

 $[x \mapsto 3] ((\lambda x.x + 1) (x + 1)) = (\lambda x.x + 1) (3 + [x \mapsto 3] 1)$ 

$$[x \mapsto s]$$
  $(t_1 \ t_2)$   $=$   $([x \mapsto s] \ t_1)$   $([x \mapsto s] \ t_2)$ 

$$\mapsto s]t_2)$$

$$t_2)$$

if 
$$y \neq x \land y \notin FV(s)$$

$$[x \mapsto s] x = s$$

$$[x \mapsto s] \ y = y$$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

$$= y$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

 $[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$ 

$$= \lambda y.([x$$

$$= ([x \mapsto s])$$

$$([x \mapsto x])$$

$$x \mapsto s \mid t$$

$$\rightarrow s$$
]  $t_1$ )

 $[x \mapsto 3]$   $((\lambda x.x + 1)$   $(x + 1)) = (\lambda x.x + 1)$   $(3 + [x \mapsto 3]$  1)

$$[x \mapsto s] (t_1 \ t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

$$x\mapsto s]t_2$$

$$s\mapsto s]t_2)$$

if 
$$y \neq x \land y \notin FV(s)$$

 $[x \mapsto s] v$ 

$$[x \mapsto s] x = s$$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

$$= y$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$= \lambda v.([x])$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

 $[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$ 

$$= \lambda y.([x$$

$$([ \lor \hookrightarrow e]$$

$$\mapsto s | t_1 )$$

$$\langle \mapsto s | t_1 \rangle$$

$$s] t_1)$$
 (

 $[x \mapsto 3]$   $((\lambda x.x + 1) (x + 1)) = (\lambda x.x + 1) (3 + [x \mapsto 3] 1)$ 

$$[x\mapsto s]$$
  $(t_1\ t_2)$   $=$   $([x\mapsto s]\ t_1)$   $([x\mapsto s]\ t_2)$ 

$$\rightarrow s]t_2)$$

$$t_2)$$

$$y \neq x \land y \notin$$

if 
$$y \neq x \land y \notin FV(s)$$

$$[x \mapsto s] \ x = s [x \mapsto s] \ y = y$$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$\lambda y$$
. ([ $x$ 

$$\lambda y.([x])$$

$$= \lambda y \cdot ([\lambda$$

 $[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$ 

$$\lambda y$$
. ([x

$$\mapsto s]t_1)$$

$$\rightarrow s]t_1)$$
 (

 $[x \mapsto 3]$   $((\lambda x.x + 1) (x + 1)) = (\lambda x.x + 1) (3 + [x \mapsto 3] 1)$ 

$$([x \mapsto s])$$

$$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

$$([x\mapsto s]\ t_2$$

$$|t_2)$$

If 
$$y \neq x \land$$

if 
$$y \neq x \land y \notin FV(s)$$

## Substitution rules $[x \mapsto s] x$ $[x \mapsto s] y$

$$x = s$$
  
 $y = y$ 

$$= y$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$= \lambda y.([x]$$

$$([x \mapsto s])$$

$$[x \mapsto s] (t_1 \ t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

$$=$$
 ([ $x \mapsto$ 

$$=$$
 ([ $x \mapsto$ 

$$=$$
 ([ $x \mapsto$ 

$$s] t_1) ([x$$

$$([x \mapsto s]$$

$$[x \mapsto s] t_1) + ([x \mapsto s] t_2)$$

$$[x \mapsto s] |\langle int \rangle = |\langle int \rangle | [x \mapsto s] |(t_1 + t_2) = ([x \mapsto s] |t_1) + ([x \mapsto s] |t_2)$$

$$([x \mapsto s] t_1)$$

$$-([x\mapsto s]\ t_2)$$

if  $y \neq x$ 

if  $y \neq x \land y \notin FV(s)$ 

$$[x \mapsto 3] ((\lambda x.x+1) (x+1)) = (\lambda x.x+1) (3 + [x \mapsto 3] 1)$$

$$(3+[x]$$

$$= \left(\lambda x.x + 1\right) \left(3 + 1\right)$$

$$3 + 1$$

#### Substitution rules $[x \mapsto s] x$ = s $[x \mapsto s] y$

$$= y$$

$$c.t_1$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$
$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] (x \mapsto s)]$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$
$$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

$$([x \mapsto s]$$

$$) = ([x \vdash$$

$$[x \mapsto s] \langle int \rangle = \langle int \rangle$$

$$= \frac{\langle int \rangle}{\langle int \rangle}$$

$$[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$$

 $[x \mapsto 3]$   $((\lambda x.x + 1)$  (x + 1)) =  $(\lambda x.x + 1)$   $(3 + [x \mapsto 3] 1)$ 

$$s] \langle int \rangle$$

 $= \left(\lambda x.x+1\right) \left(3+1\right)$ 

if 
$$y \neq x$$

if 
$$y \neq x \land y \notin FV(s)$$

$$[x \mapsto s] x$$

$$[x \mapsto s] \ x = s [x \mapsto s] \ y = y$$

 $[x \mapsto s] \langle int \rangle = \langle int \rangle$ 

$$= y$$

$$[x \mapsto s] (\lambda x.t_1) = \lambda x.t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y.([x \mapsto s] t_1)$$

$$(\lambda y.t_1) = \lambda y.($$

$$(t_1.t_2) = ([x \vdash$$

$$(x \mapsto x) = (x \mapsto x)$$

$$= ([x \mapsto s])$$

$$\lambda y.([x \mapsto s] t_1$$

$$[x \mapsto s] (\lambda y.t_1) = \lambda y. ([x \mapsto s] t_1) \quad \text{if } y \neq x \land y \notin FV(s)$$
$$[x \mapsto s] (t_1 t_2) = ([x \mapsto s] t_1) ([x \mapsto s] t_2)$$

$$t_1$$
)  $([x \mapsto s])$ 

$$([x \mapsto s] t_2)$$

$$)+([x\mapsto s]t_2)$$

$$[x \mapsto s] (t_1 + t_2) = ([x \mapsto s] t_1) + ([x \mapsto s] t_2)$$

$$[x \mapsto 3] ((\lambda x.x+1) (x+1)) = (\lambda x.x+1) (3+1)$$

$$[x \mapsto s] (\lambda y.t_1$$

$$\lambda x.t_1) = \lambda x.$$

$$= y$$
 $= \lambda \times t$ 

if 
$$y \neq x$$

# Small-step semantics (eager)

$$egin{array}{cccc} t_1 & t_2 \longrightarrow t_1' & t_2 \ \hline t_2 \longrightarrow t_2' \end{array}$$

$$\frac{t_2 \longrightarrow t_{2'}}{v_1 \ t_2 \longrightarrow v_1 \ t_2}$$

 $(\lambda x.t_{12})$   $v_2$   $\longrightarrow$   $x \mapsto v_2$   $t_{12}$ 

$$t_1 \ t_2 \longrightarrow t_1' \ t_2$$



(E-APP1)

(E-APPLAM)

## Small-step semantics (lazy)

 $\frac{1}{t_1} \xrightarrow{t_2} t_1' \xrightarrow{t_2}$ 

(E-APP1)

(E-APPLAM)

```
data Term a =
   TmVar a
   | TmApp (Term a) (Term a)
   | TmLam String (Scope () Term a)
```

deriving (Eq, Ord, Show, Functor, Foldable, Traversable)

instance Monad Term where
 return = TmVar

. . .

```
TmVar x >>= f = f x
TmApp tm1 tm2 >>= f = TmApp (tm1 >>= f) (tm2 >>= f)
TmLam v s >>= f = TmLam v (s >>>= f)
```

data Scope b f a = ...

data Scope b f a = ...

abstract1 :: (Monad f, Eq a)  $\Rightarrow$  a  $\rightarrow$  f a  $\rightarrow$  Scope () f a

data Scope b f a = ... abstract1 :: (Monad f, Eq a)  $\Rightarrow$  a  $\Rightarrow$  f a  $\Rightarrow$  Scope () f a instantiate1 :: Monad f  $\Rightarrow$  f a  $\Rightarrow$  Scope n f a  $\Rightarrow$  f a

```
> let fn = TmAdd (TmVar "x") (TmInt 1)
```

```
> let fn = TmAdd (TmVar "x") (TmInt 1)
> fn
```

```
> let fn = TmAdd (TmVar "x") (TmInt 1)
> fn
TmAdd (TmVar "x") (TmInt 1)
```

```
> let fn = TmAdd (TmVar "x") (TmInt 1)
> fn
TmAdd (TmVar "x") (TmInt 1)
> toList fn
```

```
> let fn = TmAdd (TmVar "x") (TmInt 1)
> fn
TmAdd (TmVar "x") (TmInt 1)
> toList fn
["x"]
```

```
> let fn = TmAdd (TmVar "x") (TmInt 1)
> fn
TmAdd (TmVar "x") (TmInt 1)
> toList fn
["x"]
> let s = abstract1 "x" fn
```

```
> let fn = TmAdd (TmVar "x") (TmInt 1)
> fn
TmAdd (TmVar "x") (TmInt 1)
> toList fn
["x"]
> let s = abstract1 "x" fn
> s
```

```
> let fn = TmAdd (TmVar "x") (TmInt 1)
> fn
TmAdd (TmVar "x") (TmInt 1)
> toList fn
["x"]
> let s = abstract1 "x" fn
> s
Scope (TmAdd (TmVar (B ())) (TmInt 1))
```

```
> let fn = TmAdd (TmVar "x") (TmInt 1)
> fn
TmAdd (TmVar "x") (TmInt 1)
> toList fn
["x"]
> let s = abstract1 "x" fn
> s
Scope (TmAdd (TmVar (B ())) (TmInt 1))
> toList s
```

```
> let fn = TmAdd (TmVar "x") (TmInt 1)
> fn
TmAdd (TmVar "x") (TmInt 1)
> toList fn
["x"]
> let s = abstract1 "x" fn
> s
Scope (TmAdd (TmVar (B ())) (TmInt 1))
> toList s
```

```
> let fn = TmAdd (TmVar "x") (TmInt 1)
> fn
TmAdd (TmVar "x") (TmInt 1)
> toList fn
\lceil \|x\| \rceil
> let s = abstract1 "x" fn
> s
Scope (TmAdd (TmVar (B ())) (TmInt 1))
> toList s
> instantiate1 (TmInt 3) s
```

```
> let fn = TmAdd (TmVar "x") (TmInt 1)
> fn
TmAdd (TmVar "x") (TmInt 1)
> toList fn
\lceil \|x\| \rceil
> let s = abstract1 "x" fn
> s
Scope (TmAdd (TmVar (B ())) (TmInt 1))
> toList s
> instantiate1 (TmInt 3) s
TmAdd (TmInt 3) (TmInt 1)
```

```
> let s' = abstract1 "y" fn
```

```
> let s' = abstract1 "y" fn
> s'
```

```
> let s' = abstract1 "y" fn
> s'
Scope (TmAdd (TmVar (F (TmVar "x"))) (TmInt 1))
```

```
> let s' = abstract1 "y" fn
> s'
Scope (TmAdd (TmVar (F (TmVar "x"))) (TmInt 1))
```

> toList s'

```
> let s' = abstract1 "y" fn
> s'
Scope (TmAdd (TmVar (F (TmVar "x"))) (TmInt 1))
> toList s'
["x"]
```

```
> let s' = abstract1 "y" fn
> s'
Scope (TmAdd (TmVar (F (TmVar "x"))) (TmInt 1))
> toList s'
```

> instantiate1 (TmInt 3) s'

["x"]

```
> let s' = abstract1 "y" fn
> s'
Scope (TmAdd (TmVar (F (TmVar "x"))) (TmInt 1))
> toList s'
```

> instantiate1 (TmInt 3) s'
TmAdd (TmVar "x") (TmInt 1)

["x"]

lam :: String -> Term String -> Term String
lam v tm = TmLam v (abstract1 v tm)

```
lam v tm = TmLam v (abstract1 v tm)
> let fn = lam "x" $ TmAdd (TmVar "x") (TmInt 1)
```

lam :: String -> Term String -> Term String

```
lam :: String -> Term String -> Term String
lam v tm = TmLam v (abstract1 v tm)
> let fn = lam "x" $ TmAdd (TmVar "x") (TmInt 1)
> fn
```

```
lam :: String -> Term String -> Term String
lam v tm = TmLam v (abstract1 v tm)
> let fn = lam "x" $ TmAdd (TmVar "x") (TmInt 1)
> fn
TmLam "x"
  (Scope (TmAdd (TmVar (B ())) (TmInt 1)))
```

```
lam v tm = TmLam v (abstract1 v tm)
> let fn = lam "x" $ TmAdd (TmVar "x") (TmInt 1)
> fn

TmLam "x"
  (Scope (TmAdd (TmVar (B ())) (TmInt 1)))
> let tm = TmApp fn (TmAdd (TmVar "x") (TmInt 1)
```

lam :: String -> Term String -> Term String

```
lam :: String -> Term String -> Term String
lam v tm = TmLam v (abstract1 v tm)
> let fn = lam "x" $ TmAdd (TmVar "x") (TmInt 1)
> fn
TmLam "x"
  (Scope (TmAdd (TmVar (B ())) (TmInt 1)))
> let tm = TmApp fn (TmAdd (TmVar "x") (TmInt 1)
> tm
```

```
lam :: String -> Term String -> Term String
lam v tm = TmLam v (abstract1 v tm)
> let fn = lam "x" $ TmAdd (TmVar "x") (TmInt 1)
> fn
TmLam "x"
  (Scope (TmAdd (TmVar (B ())) (TmInt 1)))
> let tm = TmApp fn (TmAdd (TmVar "x") (TmInt 1)
> tm
TmApp
  (TmLam "x" (Scope (TmAdd (TmVar (B ())) (TmInt 1))))
  (TmAdd (TmVar "x") (TmInt 1))
```

```
lam :: String -> Term String -> Term String
lam v tm = TmLam v (abstract1 v tm)
> let fn = lam "x" $ TmAdd (TmVar "x") (TmInt 1)
> fn
TmLam "x"
  (Scope (TmAdd (TmVar (B ())) (TmInt 1)))
> let tm = TmApp fn (TmAdd (TmVar "x") (TmInt 1)
> tm
TmApp
```

(TmLam "x" (Scope (TmAdd (TmVar (B ())) (TmInt 1))))

> tm >>= v -> if v == "x" then TmInt 3 else TmVar v

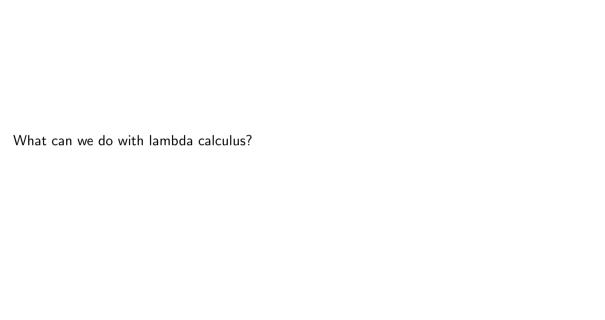
(TmAdd (TmVar "x") (TmInt 1))

```
lam :: String -> Term String -> Term String
lam v tm = TmLam v (abstract1 v tm)
> let fn = lam "x" $ TmAdd (TmVar "x") (TmInt 1)
> fn
TmLam "x"
  (Scope (TmAdd (TmVar (B ())) (TmInt 1)))
> let tm = TmApp fn (TmAdd (TmVar "x") (TmInt 1)
> tm
TmApp
  (TmLam "x" (Scope (TmAdd (TmVar (B ())) (TmInt 1))))
  (TmAdd (TmVar "x") (TmInt 1))
> tm >>= \v -> if \v == "x" then TmInt 3 else TmVar \v
TmApp
  (TmLam "x" (Scope (TmAdd (TmVar (B ())) (TmInt 1))))
  (TmAdd (TmInt 3) (TmInt 1))
```

```
eAppLam :: Rule (Term a) (Term a)
eAppLam _ (TmApp (TmLam _ s) tm) =
```

Nothing

eAppLam \_ (TmApp (TmLam \_ s) tm) = pure \$ instantiate1 tm s 
$$\frac{\lambda x.t_{12}}{t_2} t_2 \longrightarrow [x \mapsto t_2] t_{12}$$



We can do Booleans:

$$tru = \lambda \ t. \ \lambda \ f. \ t$$
 $fls = \lambda \ t. \ \lambda \ f. \ f$ 
 $and = \lambda \ b. \ \lambda \ c. \ b \ c \ fls$ 

We can do natural numbers:

$$z=\lambda\ s.\ \lambda\ z.\ z$$
  $scc=\lambda\ n.\ \lambda\ s.\ \lambda\ z.\ s\ (n\ s\ z)$  plus  $m\ n=\lambda\ s.\ \lambda\ z.\ m\ s\ (n\ s\ z)$ 

We can do pairs:

$$pair = \lambda \ f. \ \lambda \ s. \ \lambda \ b. \ b \ f \ s$$
 $fst = \lambda \ p. \ p \ tru$ 
 $snd = \lambda \ p. \ p \ fls$ 

$$fst (pair v w) \Rightarrow v$$

We even have enough to do recursion:

$$fix = \lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))$$

$$g = \lambda$$
 fct.  $\lambda$  n. if eq n 0 then 1 else times n (fct prd n)

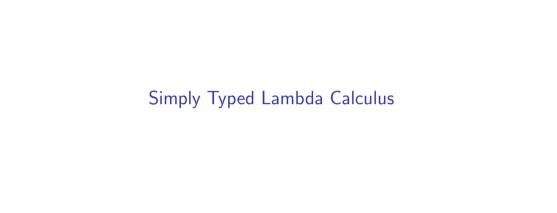
$$factorial = fix g$$

Sometimes those kind of hijinx lead us into trouble:

$$omega = (\lambda x. xx)(\lambda x. xx)$$

One other big problem - there are plenty of stuck terms:

1 2



## Terms, values and types variable X $\lambda \times : T . t$ abstraction function application t t $\lambda \times : T . t$ abstraction

function arrow

We need some extra information to make the typing rules work.

 $T \rightarrow T$ 

## Terms, values and types variable X $\lambda \times : T . t$ abstraction function application t t $\lambda \times : T . t$ abstraction

function arrow

We add *type annotations* to the variable bindings in our lambda terms.

 $T \rightarrow T$ 

## Terms, values and types variable X $\lambda \times : T . t$ abstraction function application t t $\lambda \times : T . t$ abstraction $T \rightarrow T$ function arrow

We also add an arrow type, that describes the type of functions.

## $\frac{t_2 \longrightarrow t_2'}{v_1 \ t_2 \longrightarrow v_1 \ t_2'} \tag{E-APP2}$ $\frac{(L-APP2)}{(\lambda \times T.t_1)t_2 \longrightarrow [x \mapsto t_2] \ t_1}$

 $\frac{t_1 \longrightarrow t_1'}{t_1 \ t_2 \longrightarrow t_1' \ t_2}$ 

(E-APP1)

Small-step semantics (eager)

Unsurprisingly, the small-step semantics don't change (except we now have type annotations).

	$I \vdash \iota_1 \iota_2 : I_2$
	$\Gamma, x: T_1 \vdash t: T_2$
	$\overline{\Gamma \vdash (\lambda \times : T_1.t) : T_1 \rightarrow T_2}$
Now we need a context for our tyr	ning rules

Typing rules

 $\Gamma \vdash t_1:T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2:T_1$  $\Gamma \vdash t_1 \ t_2 : T_2$ 

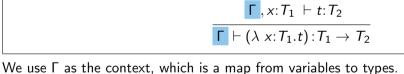
 $\Gamma$ ,  $x:T \vdash x:T$ 

 $x:T_1 \vdash t:T_2$ 

(T-ABS)

(T-VAR)

(T-App)



Typing rules

 $\Gamma$ ,  $x:T \vdash x:T$  $\Gamma \vdash t_1: \overline{T_1} \rightarrow T_2$   $\Gamma \vdash t_2: T_1$ (T-App) $\Gamma \vdash t_1 \ t_2 : T_2$ 

(T-VAR)

(T-ABS)

1, x.t + x.t	
$\Gamma \vdash t_1:T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2:T_1$	(T-App)
$\Gamma \vdash t_1 \ t_2 : T_2$	(I-AFF)
$\Gamma, x: T_1 \vdash t: T_2$	(TD A)
$\Gamma \vdash (\lambda \times : T_1.t) : T_1 \to T_2$	(T-Abs)

 $\Gamma \quad \vee \cdot T \quad \vdash \quad \vee \cdot T$ 

(T-VAR)

T-Var just grabs the type from the context.

**Typing rules** 

$\Gamma, x: T_1 \vdash t: T_2$
$\Gamma \vdash (\lambda x: T_1.t): T_1 \to T_2$

Typing rules

 $\Gamma$ ,  $x:T \vdash x:T$  $\Gamma \vdash t_1:T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2:T_1$ 

 $\Gamma \vdash t_1 \ t_2 : T_2$ 

A type error occurs if the variable isn't found in the context.

(T-App)

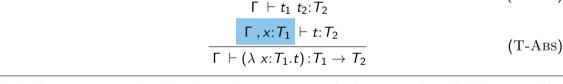
(T-VAR)

(T-ABS)

$\overline{\Gamma, x: T \vdash x: T}$	(T-VAR)
$rac{\Gamma \;dash t_1{:} T_1  ightarrow  T_2 \qquad \Gamma \;dash t_2{:} T_1}{\Gamma \;dash t_1 \;t_2{:} T_2}$	(T-App)
$rac{\Gamma \ , x  m{:} T_1 \ dash t  m{:} T_2}{\Gamma \ dash (\lambda \ x  m{:} T_1.t)  m{:} T_1  ightarrow T_2}$	(T-Abs)

T-App has no new techniques in it.

Typing rules



 $\Gamma, x: T \vdash x: T$   $\Gamma \vdash t_1: T_1 \to T_2 \qquad \Gamma \vdash t_2: T_1$ 

(T-VAR)

(T-App)

Typing rules

In T-Abs we temporarily add  $x:T_1$  to the context, just for long enough to find the type of t.

1, 3.7 + 3.7	
$\Gamma \; dash \; t_1{:}T_1  ightarrow T_2 \qquad \Gamma \; dash \; t_2{:}T_1$	(T-App)
$\Gamma \vdash t_1 \ t_2 : T_2$	(1-APP)
$\Gamma, x: T_1 \vdash t: T_2$	(/ (
$\overline{ \Gamma \ \vdash (\lambda \ x : T_1.t) : T_1 \rightarrow T_2 }$	(T-Abs)
	$egin{array}{cccccccccccccccccccccccccccccccccccc$

Typing rules

If we didn't modify the context then we would risk a type error occurring if the variable x appeared within the term t.

 $\Gamma \quad \mathbf{v} \cdot T \quad \vdash \mathbf{v} \cdot T$ 

(T-VAR)

$\frac{\Gamma_{\cdot} x : T_{1} \vdash t : T_{2}}{\Gamma_{\cdot} \vdash (\lambda_{\cdot} x : T_{1}.t) : T_{1} \rightarrow T_{2}}$	(T-Abs)
With that done we know the type of the argument and of the result, so we are d	one.

 $\Gamma$ ,  $x:T \vdash x:T$  $\Gamma \vdash t_1:T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2:T_1$ 

 $\Gamma \vdash t_1 \ t_2 : T_2$ 

(T-VAR)

(T-App)

**Typing rules** 

 $\emptyset \vdash (\lambda x : \mathsf{Bool}.x) \mathsf{true} : \mathsf{Bool}$ 

– T-Abs \_\_\_\_\_\_ T-True  $\emptyset \vdash \mathsf{true} : \mathsf{Bool}$ -T-App

 $\emptyset \vdash (\lambda x : \mathsf{Bool}.x) \mathsf{true} : \mathsf{Bool}$ 

— T-Abs \_\_\_\_\_\_ T-True

-T-App

 $\emptyset \vdash (\lambda x : Bool.x)$  true : Bool

– T-Abs \_\_\_\_\_\_ T-True

$$x : \mathsf{Bool} \vdash x : \mathsf{Bool}$$

-T-App

– T-Abs \_\_\_\_\_\_ T-True  $\emptyset \vdash \lambda \ x : \mathsf{Bool} \ . \ x : \mathsf{Bool} \ \to \ \mathsf{Bool}$   $\emptyset \vdash \mathsf{true} : \mathsf{Bool}$ 

 $\emptyset \vdash (\lambda x : Bool.x)$  true : Bool

-T-App

– T-Abs \_\_\_\_\_\_ T-True  $\emptyset \vdash \lambda \times : \mathsf{Bool} \cdot \times : \mathsf{Bool} \rightarrow \mathsf{Bool} \qquad \emptyset \vdash \mathsf{true} : \mathsf{Bool}$ 

 $\emptyset \vdash (\lambda x: Bool. x)$  true : Bool

$$x : \mathsf{Bool} \vdash x : \mathsf{Bool}$$

 $\emptyset \vdash (\lambda x: Bool. x)$  true : Bool

– T-Abs \_\_\_\_\_\_ T-True  $\emptyset \vdash \lambda \times : \mathsf{Bool} \cdot \times : \mathsf{Bool} \to \mathsf{Bool} \qquad \emptyset \vdash \mathsf{true} : \mathsf{Bool}$ 

 $\emptyset \vdash (\lambda x : \mathsf{Bool}.x) \mathsf{true} : \mathsf{Bool}$ 

– T-Abs \_\_\_\_\_\_ T-True

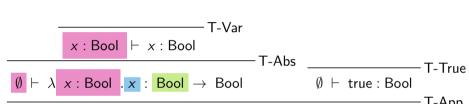
$$x : \mathsf{Bool} \vdash x : \mathsf{Bool}$$

– T-Abs \_\_\_\_\_\_ T-True

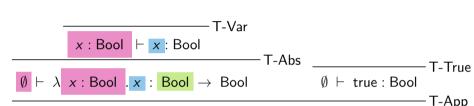
-T-App

 $\emptyset \vdash \lambda x : \mathsf{Bool} \ .x : \mathsf{Bool} \to \mathsf{Bool}$   $\emptyset \vdash \mathsf{true} : \mathsf{Bool}$ 

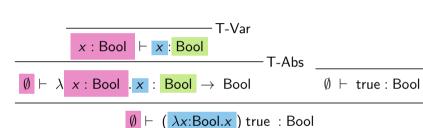
 $\emptyset \vdash (\lambda x : Bool.x)$  true : Bool

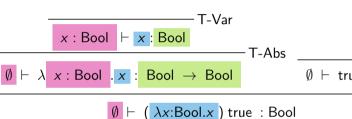


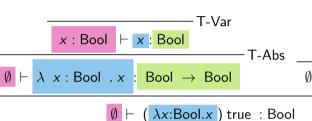
 $\emptyset \vdash (\lambda x : Bool.x)$  true : Bool

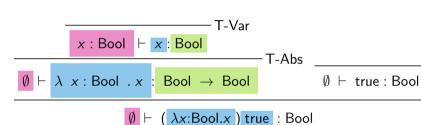


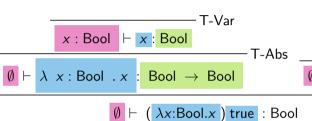
 $\emptyset \vdash (\lambda x : Bool.x)$  true : Bool



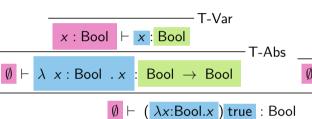




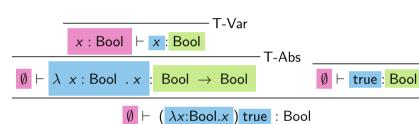


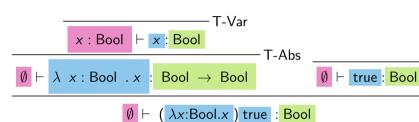


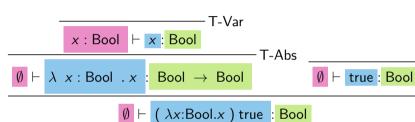
**Ø** ⊢ true : Bool



\_\_\_\_\_ T-True <mark>∅ ⊢ true</mark> : Bool







```
inferVar :: Ord a => Rule (Context a, Term a) Type
inferVar _ (ctx, TmVar a) =
fetchFromContext a ctx
inferVar _ =
```

Nothing

(T-VAR)

```
inferApp :: Rule (Context a, Term a) Type
inferApp step (ctx, TmApp tm1 tm2) = do
  ty1 <- step (ctx, tm1)
  ty2 <- step (ctx, tm2)
  case ty1 of
    TyArr tyF tyT ->
    if ty2 == tyF
    then pure tyT
    else Nothing
```

\_ -> Nothing

inferApp \_ \_ =
 Nothing

```
\frac{\Gamma \vdash t_1: T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2: T_1}{\Gamma \vdash t_1 \ t_2: T_2} (\text{T-App})
```

```
inferLam step (ctx, TmLam v ty s) = do
```

tyT <- step ( addToContext v ty ctx , instantiate1 (TmVar v) s

pure \$ TyArr ty tyT

inferLam \_ = Nothing

 $\frac{\Gamma, x: T_1 \vdash t: T_2}{\Gamma \vdash (\lambda x: T_1.t): T_1 \to T_2}$  (T-Abs)

inferLam :: Rule (Context String, Term String) Type

```
checkVar :: Ord a => Rule (Context a, Term a, Type) ()
checkVar _ (ctx, TmVar v, ty) = do
  tyC <- fetchFromContext v ctx</pre>
                                                          \Gamma, x:T \vdash x:T
```

if tyC == ty then pure () else Nothing

checkVar \_ \_ = Nothing

(T-VAR)

```
checkApp :: RuleSet (Context String, Term String) Type
           -> Rule (Context String, Term String, Type) ()
checkApp infer step (ctx, TmApp tm1 tm2, ty) = do
  ty1 <- infer (ctx, tm1)
  case ty1 of
     TyArr tyF tyT -> do
                                                         \frac{\Gamma \vdash t_1: T_1 \to T_2 \qquad \Gamma \vdash t_2: T_1}{\Gamma \vdash t_1 \ t_2: T_2} (\text{T-APP})
        if tvT == tv
```

then step (ctx, tm2, tyF)

else Nothing

Nothing checkApp \_ \_ = Nothing

->

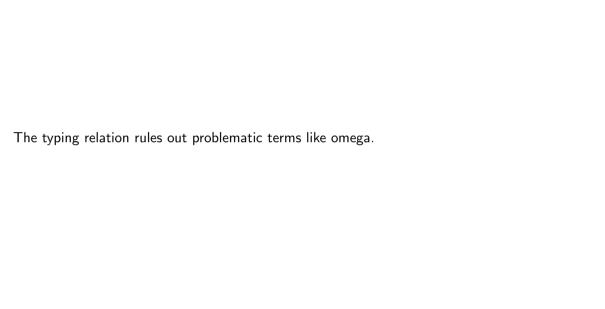
```
checkLam :: Rule (Context String, Term String, Type) ()
checkLam step (ctx, TmLam v ty s, TyArr tyF tyT) = do
```

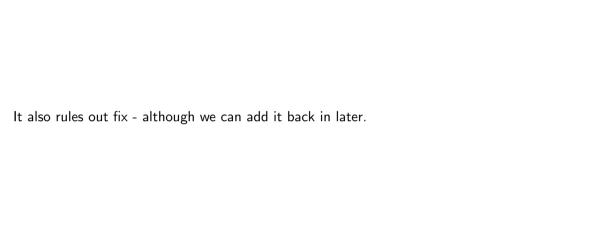
```
if ty == tyF
then do
```

step ( addToContext v ty ctx , instantiate1 (TmVar v) s , tyT

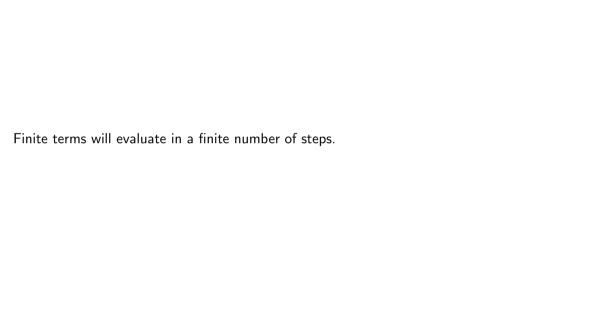
else Nothing checkLam = Nothing

 $\frac{\Gamma, x: T_1 \vdash t: T_2}{\Gamma \vdash (\lambda x: T_1.t): T_1 \to T_2}$  (T-Abs)





t rules out enough problematic terms that STLC is actually strongly normalizing.	





#### Terms

$$t := \dots$$

fix t fixed point

### Small-step semantics

$$\frac{t \longrightarrow t'}{\text{fix } t \longrightarrow \text{fix}}$$

 $\overline{\text{fix}(\lambda \times : T.t)} \longrightarrow [x \mapsto \text{fix}(\lambda \times : T.t)] t$ 

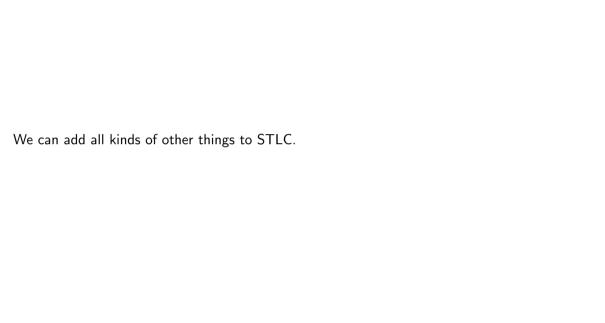
(E-Fix1)

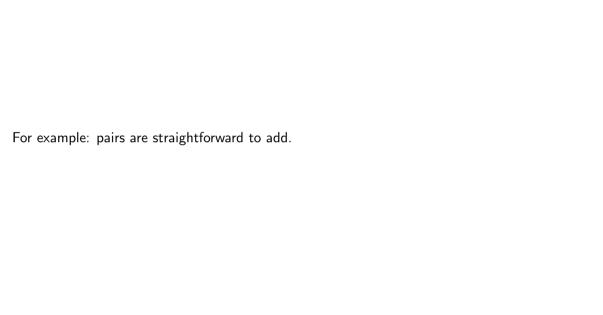
(E-FIXBETA)

### Typing rules

$$\frac{\vdash t: T \to T}{\vdash \textit{fix } t: T}$$

(T-Fix)





## Terms, values and types

(t,t)

fst t

snd t

(v, v)

 $T \times T$ 

pair elimination

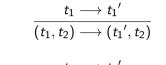
pair elimination

pair type

pair value

pair introduction

## **Small-step semantics**



 $snd(v_1, v_2) \longrightarrow v_2$ 

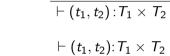
$$rac{t_2 \longrightarrow t_2'}{(v_1,t_2) \longrightarrow (v_1,t_2')}$$

$$\overline{\mathit{fst}\left(\mathit{v}_{1},\mathit{v}_{2}
ight)\longrightarrow\mathit{v}_{1}}$$

(E-Pair1)

(E-Pair2)

# Typing rules



 $\vdash t_1:T_1 \qquad \vdash t_2:T_2$ 

 $\vdash$  fst  $\overline{(t_1,t_2):T_1}$ 

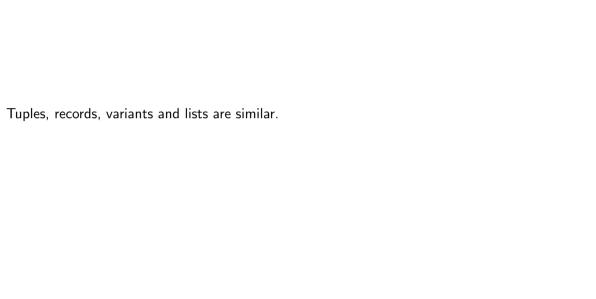
 $\vdash (t_1, t_2): T_1 \times T_2$ 

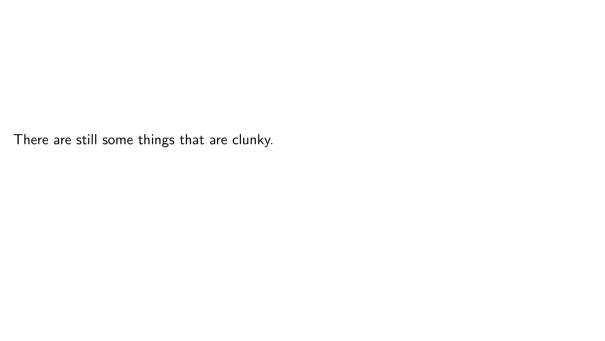
 $\vdash$  snd  $(t_1, t_2): T_2$ 

(T-PairFst)

(T-Pair)

(T-PairSnd)





We have to write a lot of different versions of id

$$\lambda x$$
: Bool .  $x$ 

$$\lambda x$$
: Int .  $x$ 

Things are worse for const

$$\lambda \ x$$
 : Bool .  $\lambda \ y$  : Bool .  $x$ 

$$\lambda x : \mathsf{Bool} \cdot \lambda y : \mathsf{Int} \cdot x$$
  
 $\lambda x : \mathsf{Int} \cdot \lambda y : \mathsf{Bool} \cdot x$ 

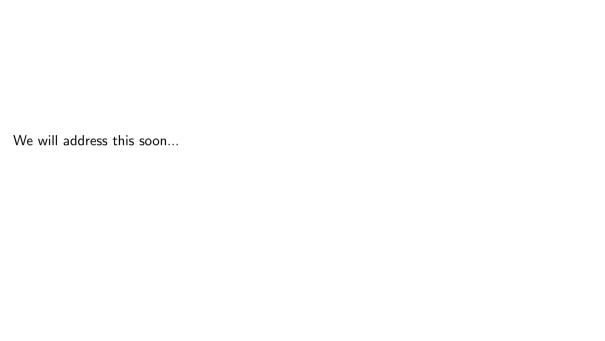
$$\lambda x : \mathsf{Int} \cdot \lambda y : \mathsf{Bool} \cdot x$$

 $\lambda x : Int . \lambda y : Int . x$ 

Don't even get me started on compose

$$\lambda \ f : \mathsf{Bool} \to \mathsf{Int} \ . \ \lambda \ g : \mathsf{Int} \to \mathsf{Bool} \ . \ \lambda \ x : \mathsf{Int} \ . \ f \ (g \ x)$$

. . .



Next time...

▶ Pattern matching

- ▶ Pattern matching
- ► Recursive types

- ► Pattern matching
- Recursive types
- User defined data types

- Pattern matching
- Recursive types
- User defined data types
- ► Hindley Milner

- ► Pattern matching
- Recursive types
- User defined data types
- Hindley Milner

System F

- ► Pattern matching
- Recursive types
- User defined data types
- Hindley Milner
- System F

ightharpoonup System F $\omega$ 

Code and slides are in version-2 at	
https://github.com/dalaing/plt-talk	