#### Programming Language Theory

Dave Laing

BFPG March 2017

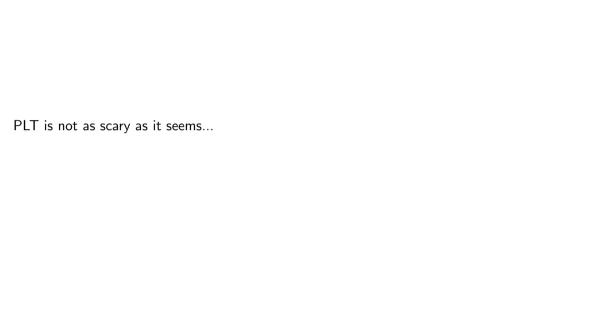


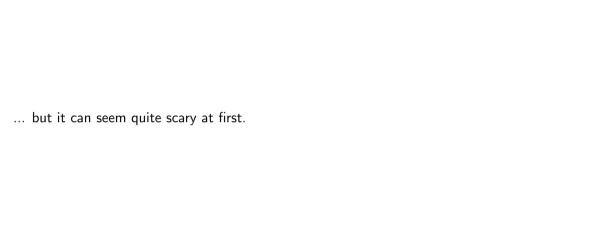
Programming Language Theory (PLT) is handy to know about.	

Useful for understanding how different languages.	guages work,	comparing them,	coming up	with new

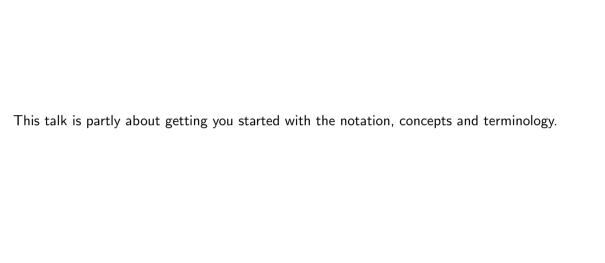


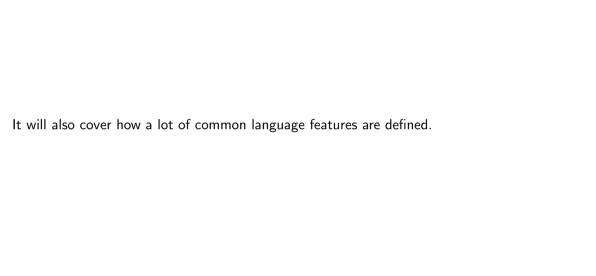
You only need to know about a few things before you can read <i>many</i> mor	e papers.



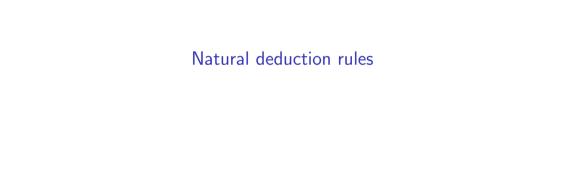


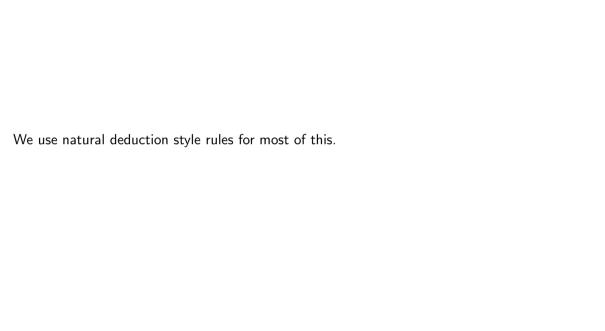
Terms, values and types	Small-step semantics
$egin{array}{cccc} t & := & & & & & & & & & & & & & & & & & $	$rac{t_1 \longrightarrow t_1{}'}{t_1 \ t_2 \longrightarrow t_1{}' \ t_2}$ (E-App1)
$t \ t$ function application $v := \lambda \ x : T . t$ abstraction	$\frac{t_2 \longrightarrow t_2'}{v_1 \ t_2 \longrightarrow v_1 \ t_2'} \tag{E-App2}$
T := $T  o T$ function arrow	$\overline{(\lambda x: T.t_1)t_2 \longrightarrow [x \mapsto t_2] t_1}  \text{(E-AppAbs)}$
	Typing rules
	$\frac{x:T\in\Gamma}{\Gamma\vdash x:T} \tag{T-Var}$
	$\frac{\Gamma \vdash t_1: T_1 \to T_2 \qquad \Gamma \vdash t_2: T_1}{\Gamma \vdash t_1 \ t_2: T_2} \tag{T-APP}$
	$\frac{\Gamma, x: \mathcal{T}_1 \vdash \mathcal{T}_2}{\Gamma \vdash (\lambda x: \mathcal{T}_1.t): \mathcal{T}_1 \to \mathcal{T}_2} $ (T-Abs)





It won't cover how to prove these various properties, although "Types and Programming Languages" and/or "Software Foundations" cover that very well if you are interested in that.





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

 $Assumption_1$ 

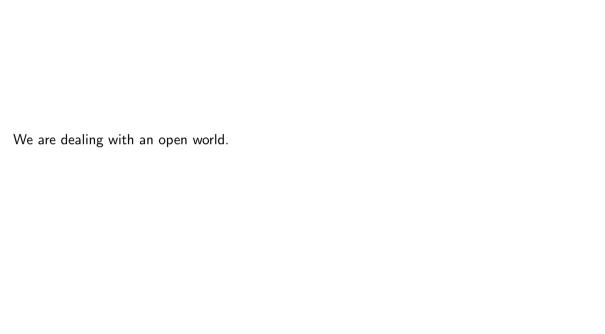
Conclusion

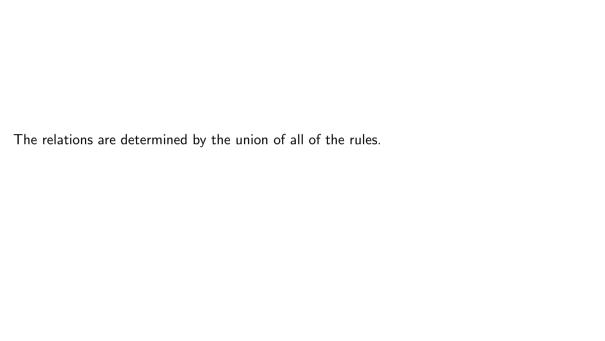
### $\frac{}{\text{even 0}}$ (EVEN-ZERO)

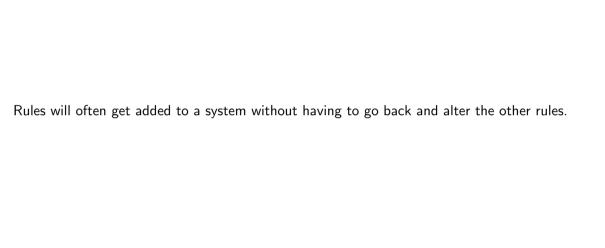
(EVEN-ADD)

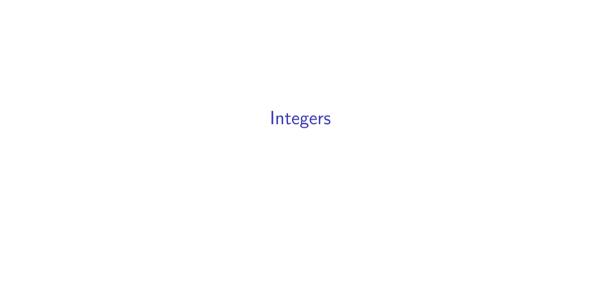
even x

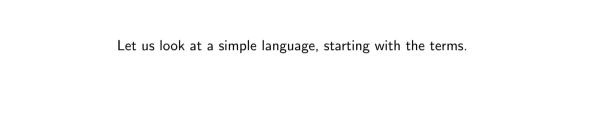
even (x+2)





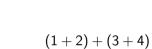


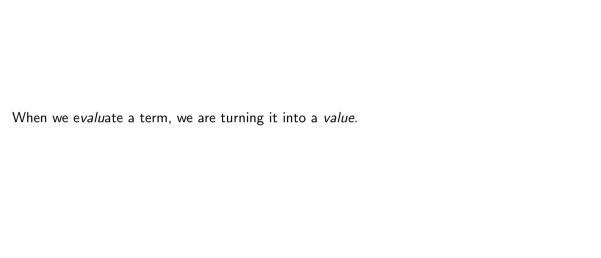


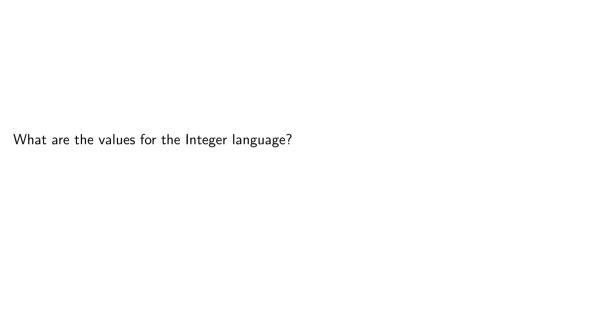


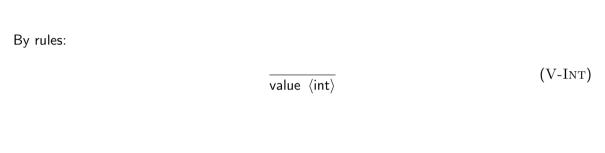
```
\langle \mathsf{int} 
angle \hspace{0.5cm} \textit{constant integer} \ t+t \hspace{0.5cm} \textit{addition}
```











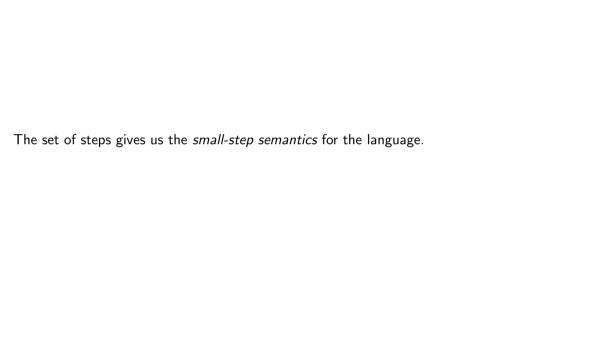
By definitions:

:=

.—

⟨int⟩ constant integer





## $\frac{t_1 \longrightarrow t_1'}{t_1 + t_2 \longrightarrow t_1' + t_2}$ $\frac{t_2 \longrightarrow t_2'}{v_1 + t_2 \longrightarrow v_1 + t_2'}$

**Small-step semantics** 

(E-ADD2)

(E-ADD1)

(E-AddInt)

The steps are specified as a binary relation  $t_1 \longrightarrow t_2$ .

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

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

(E-ADD1)

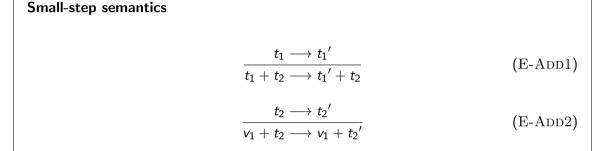
(E-ADD2)

(E-AddInt)

**Small-step semantics** 

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

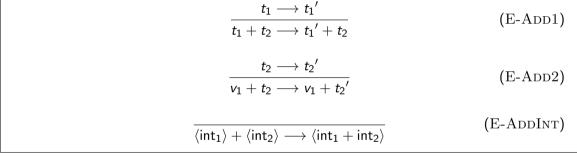
 $\langle \mathsf{int}_1 \rangle + \langle \mathsf{int}_2 \rangle \longrightarrow \langle \mathsf{int}_1 + \mathsf{int}_2 \rangle$ 



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

(E-AddInt)

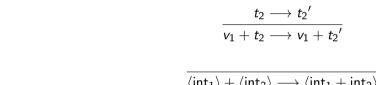
E-AddInt does the actual addition.



(E-ADD1)

**Small-step semantics** 

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



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

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

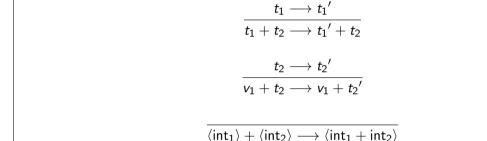
**Small-step semantics** 

(E-ADD2)

(E-ADD1)

(E-AddInt)

Any term that cannot take a step is known as a normal form.



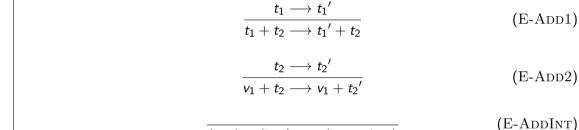
(E-ADD1)

(E-ADD2)

(E-AddInt)

**Small-step semantics** 

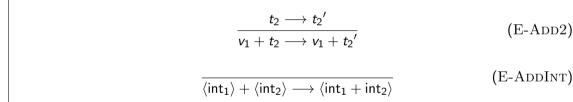
Values cannot take a step by definition and so are always normal forms.



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

**Small-step semantics** 

Iterating the small-step relation until you reach a value is called *evaluation*.

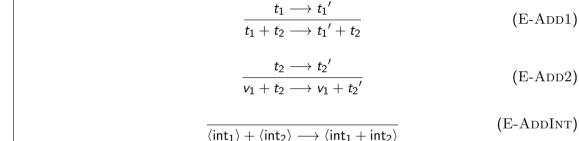


 $\frac{t_1 \longrightarrow t_1'}{t_1 + t_2 \longrightarrow t_1' + t_2}$ 

(E-ADD1)

**Small-step semantics** 

Iterating the small-step relation until you reach a normal form is called *normalization*.



(E-ADD1)

**Small-step semantics** 

Usually evaluation and normalization are / are hoped to be the same thing.

## $\frac{t_2 \longrightarrow t_2'}{v_1 + t_2 \longrightarrow v_1 + t_2'}$ $\langle \mathsf{int}_1 \rangle + \langle \mathsf{int}_2 \rangle \longrightarrow \langle \mathsf{int}_1 + \mathsf{int}_2 \rangle$

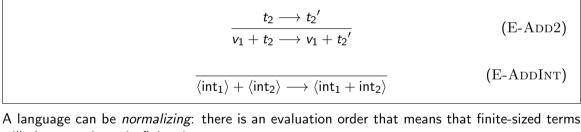
 $\frac{t_1 \longrightarrow t_1'}{t_1 + t_2 \longrightarrow t_1' + t_2}$ 

If a term is not a value but is a normal form, then it is stuck.

**Small-step semantics** 

(E-ADD1)

(E-ADD2)

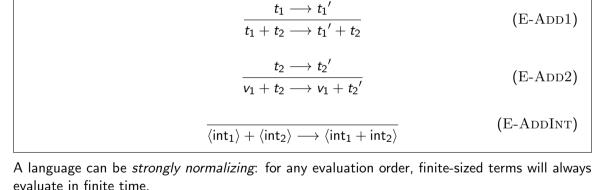


 $\frac{t_1 \longrightarrow t_1'}{t_1 + t_2 \longrightarrow t_1' + t_2}$ 

(E-ADD1)

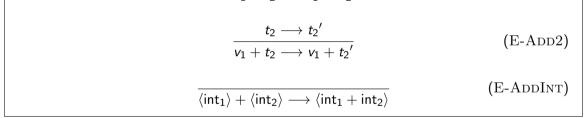
**Small-step semantics** 

will always evaluate in finite time.



(E-ADD1)

**Small-step semantics** 



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

(E-ADD1)

**Small-step semantics** 

The relationship between values and normal forms is a relationship between syntax and semantics.

We can define evaluation in terms of the small-step relation:

$\overline{v \Rightarrow v}$	(Big-Value)
- , -	

$$t\longrightarrow t' \qquad t'\Rightarrow v$$

$$\frac{t \longrightarrow t' \qquad t' \Rightarrow v}{t \longrightarrow t' \longrightarrow v} \tag{Big-Step}$$

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

$$\dfrac{\overline{\langle \mathsf{int}_1 \rangle} \ + \ \langle \mathsf{int}_2 \rangle \ \longrightarrow \ \langle \mathsf{int}_1 + \mathsf{int}_2 \rangle}{1 + 2 \ \longrightarrow \ 3} \dfrac{\mathsf{E} ext{-}\mathsf{AddInt}}{\Big(1 + 2\Big) + \Big(3 + 4\Big) \ \longrightarrow \ 3 + \Big(3 + 4\Big)} \mathsf{E} ext{-}\mathsf{Add1}$$

 $t_1 \longrightarrow t_1'$ 

 $t_1 + t_2 \longrightarrow t_1' + t_2$ 

(E-ADD1)

$$\frac{\langle \mathsf{int}_1 \rangle \ + \ \langle \mathsf{int}_2 \rangle \ \longrightarrow \ \langle \mathsf{int}_1 + \mathsf{int}_2 \rangle}{1 \ + \ 2 \ \longrightarrow \ 3} \mathsf{E-AddInt}$$

 $(1+2)+(3+4)\longrightarrow 3+(3+4)$ 

- E-Add1

 $t_1 \longrightarrow t_1'$ 

 $t_1 + t_2 \longrightarrow t_1' + t_2$ 

 $1 + 2 \longrightarrow 3$ 

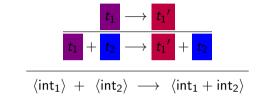
$$\frac{\phantom{-}}{1 + 2 \longrightarrow 3} \text{E-AddInt}$$

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

(E-ADD1)

$$\frac{1 + 2 \longrightarrow 3}{1 + 4} \xrightarrow{\text{E-AddInt}} \text{E-Add}$$

$$\left(1 + 2\right) + \left(3 + 4\right) \longrightarrow 3 + \left(3 + 4\right)$$



$$1+2 \longrightarrow 3$$
 E-AddInt  $(1+2)+(3+4) \longrightarrow 3+(3+4)$  E-Add

 $t_1 \longrightarrow t_1'$ 

 $t_1 + t_2 \longrightarrow t_1' + t_2$ 

(E-ADD1)

 $t_1 \longrightarrow t_1'$ 

 $t_1 + t_2 \longrightarrow t_1' + t_2$ 

(E-ADD1)

$$\frac{\langle \mathsf{int_1} \rangle}{1 + 2} \longrightarrow \frac{\langle \mathsf{int_1} + \mathsf{int_2} \rangle}{2}$$

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

 $t_1 \longrightarrow t_1'$ 

 $t_1 + t_2 \longrightarrow t_1' + t_2$ 

The complete evaluation takes three steps.

First:

Then:

Finally:

$$\frac{1}{1+2\longrightarrow 3}$$
 E-AddInt

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

 $\frac{}{3+7 \longrightarrow 10}$  E-AddInt

$$\frac{1+2-}{(1+2)+(3+4)-}$$

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

$$1+2 \longrightarrow$$

$$\frac{3}{3}$$
  $\frac{3}{3}$   $\frac{1}{3}$   $\frac{3}{3}$   $\frac{1}{3}$   $\frac{3}{3}$   $\frac{1}{3}$ 

$$\frac{3}{3+(3+4)}$$
 E-Add1

$$(3+4)$$

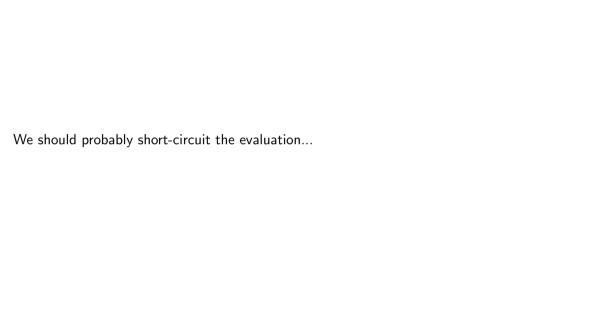
#### Booleans

# Terms and values $t := \\ false & constant \ false \\ true & constant \ true \\ t \ or \ t & disjunction \\ v := \\ false & false \ value \\ \end{cases}$

true

true value

### **Small-step semantics** $rac{t_1 \longrightarrow {t_1}'}{t_1 \; ext{or} \; t_2 \longrightarrow {t_1}' \; ext{or} \; t_2}$ (E-O<sub>R</sub>1) $\frac{t_2 \longrightarrow t_2'}{v_1 \text{ or } t_2 \longrightarrow v_1 \text{ or } t_2'}$ (E-O<sub>R</sub>2) (E-Orfalsefalse) false or false $\longrightarrow$ false (E-OrfalseTrue) false or true $\longrightarrow$ true (E-Ortruefalse) true or false $\longrightarrow$ true (E-OrtrueTrue) true or true $\longrightarrow$ true



## Small-step semantics

$$rac{t_1 \longrightarrow t_1{}'}{t_1 \; ext{or} \; t_2 \longrightarrow t_1{}' \; ext{or} \; t_2}$$

$$\frac{}{\text{false or } t_2 \longrightarrow t_2}$$

true or  $t_2 \longrightarrow true$ 

(E-Or1)

# $rac{t_1 \longrightarrow t_1'}{t_1 ext{ or } t_2 \longrightarrow t_1' ext{ or } t_2}$ (E-Or1) $\overline{ ext{false or } t_2 \longrightarrow t_2}$ (E-OrFalse) $\overline{ ext{true or } t_2 \longrightarrow ext{true}}$

In the languages we've seen so far, at most one rule applies to each term.

**Small-step semantics** 

	$\dfrac{\iota_1 \longrightarrow \iota_1}{t_1  ext{ or } t_2 \longrightarrow t_1'  ext{ or } t_2}$	(E-Or1)
	$\overline{false} \ or \ t_2 \longrightarrow t_2$	(E-Orfalse)
	$\overline{ ext{true or } t_2 \longrightarrow  ext{true}}$	(E-OrTrue)
If the rules for the steps over all of the overlapping rules, t	rlap for a particular term, but the resul	t of the step is the same for

 $t_1 \longrightarrow {t_1}'$ 

**Small-step semantics** 

### **Small-step semantics** $rac{t_1 \longrightarrow t_1'}{t_1 ext{ or } t_2 \longrightarrow t_1' ext{ or } t_2}$ (E-O<sub>R</sub>1) (E-Orfalse) false or $t_2 \longrightarrow t_2$ (E-OrTrue) true or $t_2 \longrightarrow \text{true}$

This means that we don't have to worry about the order in which the rules are applied.

## $rac{t_1 \longrightarrow t_1'}{t_1 ext{ or } t_2 \longrightarrow t_1' ext{ or } t_2}$ (E-Orfalse) $\overline{ ext{false or } t_2 \longrightarrow t_2}$ (E-Orfalse)

In other cases where rules overlap, they can be harder to deal with.

**Small-step semantics** 

	$\overline{t_1  { m or}  t_2 \longrightarrow {t_1}'  { m or}  t_2}$	(E-Or1)
	$\overline{false} \ or \ t_2 \longrightarrow t_2$	(E-Orfalse)
	$\overline{true\;or\;t_2\longrightarrowtrue}$	(E-OrTrue)
If we had the short-circui we would have some trou	ting rules and the non-short circuiting rul	es in use at the same time,

 $t_1 \longrightarrow {t_1}'$ 

**Small-step semantics** 

	$\dfrac{t_1 \  \   r \  \   t_1}{t_1 \    ext{or} \   t_2 \longrightarrow t_1' \    ext{or} \   t_2}$	(E-Or1)
	$\overline{false} \ or \ t_2 \longrightarrow t_2$	(E-Orfalse)
	$\overline{true\;or\;t_2\longrightarrowtrue}$	(E-OrTrue)
If we non-deterministically c	noose a rule to apply we will end up step	pping to different terms, but

 $t_1 \longrightarrow t_1'$ 

**Small-step semantics** 

evaluation will end up at the same value.

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

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

Things could be worse.

(EQ-OR)	$\frac{t_1 = t_2}{t_1 \text{ or } t_3 = t_2 \text{ or } t_4}$	(EQ-Refl)	$\overline{t=t}$	
(EQ-ORFALSE)	$\overline{false} \; or \; t = t$	(EQ-SYM)	$\frac{t_2=t_1}{t_1=t_2}$	
(Eq-OrTrue)	${true\;or\;t=true}$	(Eq-Trans)	$\frac{t_1=t_2\qquad t_2=t_3}{t_1=t_2}$	

Some rules have multiple valid choices for some terms that can create loops.

A non-deterministic set of rules

$\overline{t=t}$	(EQ-REFL)	$rac{t_1 = t_2}{t_1  ext{ or } t_3 = t_4}$ (EQ-OR)
$\frac{t_2=t_1}{t_1=t_2}$	(EQ-SYM)	${\text{false or } t = t}$ (EQ-ORFALSE)
$\frac{t_1 = t_2 \qquad t_2 = t_3}{t_1 = t_3}$	(Eq-Trans)	${true\;or\;t=true}$ (EQ-ORTRUE)
These are hard to implement since you	sould get study applying	

over and over.

A non-deterministic set of rules

These are hard to implement, since you could get stuck applying  $\frac{t_2=t_1}{t_1=t_2}$ 

(EQ-SYM)

t = t	,	$\frac{t_1 - t_2 - t_3}{t_1 \text{ or } t_3 = t_2 \text{ or } t_4}$	(Eq-Or)
$\frac{t_2=t_1}{t_1=t_2}$	(EQ-SYM)	$\overline{false\;or\;t=t}$	(Eq-OrFalse)
$\frac{t_1 = t_2 \qquad t_2 = t_3}{t_1 = t_3}$	(EQ-Trans)	$\overline{true}\;or\;t=true$	(Eq-OrTrue)

 $t_1 = t_2$   $t_3 = t_4$ 

(Eq-Refl)

Normally the non-deterministic rules are there to make propositions or proofs easier to work with.

A non-deterministic set of rules

$\frac{t_2=t_1}{t_1=t_2}$	(EQ-SYM)	$\overline{false}\;or\;t=t$	(Eq-OrFalse)
$rac{t_1 = t_2 \qquad t_2 = t_3}{t_1 = t_3}$	(Eq-Trans)	$\overline{true}\;or\;t=true$	(EQ-OrTrue)

(EQ-OR)

(Eq-Refl)

A non-deterministic set of rules

 $\overline{t=t}$ 

Most of the time there will also be a corresponding set of deterministic rules - often referred to as *algorithmic* - which will aid the implementors.

(EQ-OR)	$rac{t_1 = t_2 \qquad t_3 = t_4}{t_1  ext{ or } t_3 = t_2  ext{ or } t_4}$	(EQ-REFL)	$\overline{t=t}$
(Eq-OrFalse)	$\overline{false} \; or \; t = t$	(EQ-SYM)	$\frac{t_2=t_1}{t_1=t_2}$
(Eq-OrTrue)	$\frac{1}{\text{true or } t = \text{true}}$	(Eq-Trans)	$\frac{t_1 = t_2 \qquad t_2 = t_3}{t_1 = t_3}$

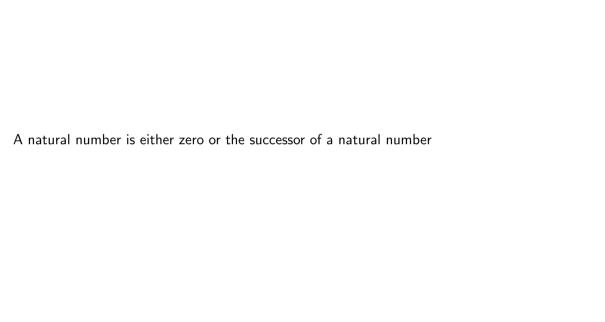
 $t_1 = t_2$   $t_3 = t_4$ 

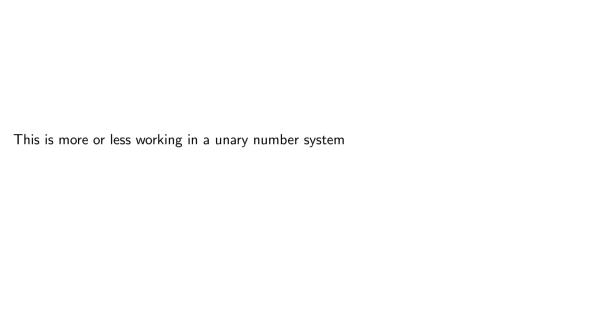
This is usually follow by a proof of equivalence between the non-deterministic and algorithmic rule sets.

(Eq-Refl)

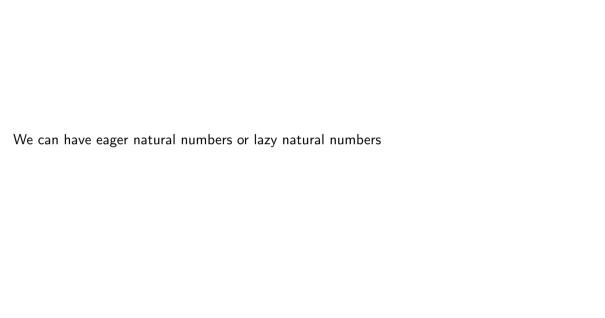
A non-deterministic set of rules







 $3 \equiv \mathit{succ} \ \mathit{succ} \ \mathit{Succ} \ \mathit{O}$ 



### 

successor value

succ *v* 

# Terms and values (lazy) $t := \\ & O \qquad constant \ zero \\ & succ \ t \qquad successor \\ & pred \ t \qquad predecessor \\ \\ \end{array}$

succ t

zero value

successor value

## Small-step semantics

$$\frac{}{\mathsf{pred} \ \mathsf{O} \longrightarrow \mathsf{O}} \tag{E-PredZero}$$

$$\frac{}{\mathsf{pred} \ (\mathsf{succ} \ v) \longrightarrow v}$$

\* Only for eager evaluation \*\* Uses v for eager evaluation and t for lazy evaluation

 $\frac{t_1 \longrightarrow t_1'}{\mathsf{succ}\ t_1 \longrightarrow \mathsf{succ}\ t_1'}$ 

 $\frac{t_1 \longrightarrow t_1'}{\mathsf{pred} \ t_1 \longrightarrow \mathsf{pred} \ t_1'}$ 

(E-Succ<sup>\*</sup>)

(E-Pred)

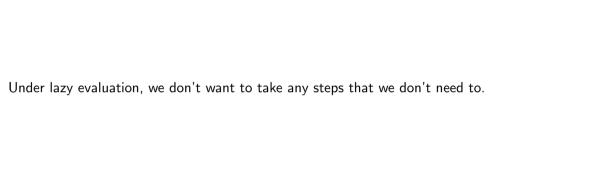
These are the same natural numbers under eager evaluation:

succ O

succ (pred (succ O))

and under lazy evaluation:

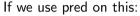
Under eager evaluation, we don't want our values to have anything in them that needs to take a step.



This is fine:

succ (pred (succ *O*))

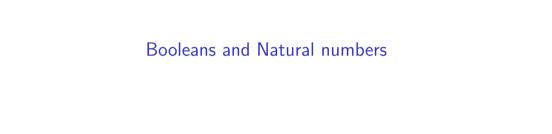
because a natural number is either zero or the successor of a natural number.

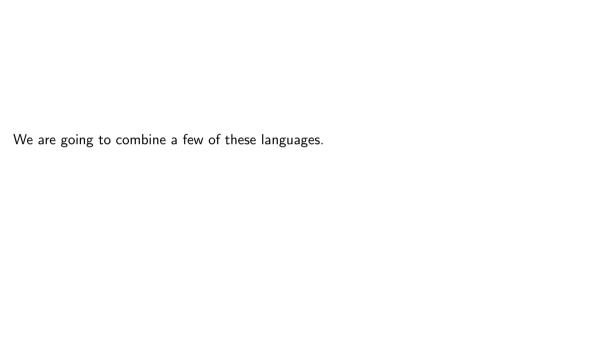


with another succ on the outside.

succ (pred (succ *O*))

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





#### 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

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

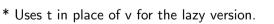
(E-IsZero)

(E-IsZeroZero)

(E-IsZeroSucc\*)

iszero 
$$O \longrightarrow true$$

iszero (succ 
$$v$$
)  $\longrightarrow$  false



# 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)

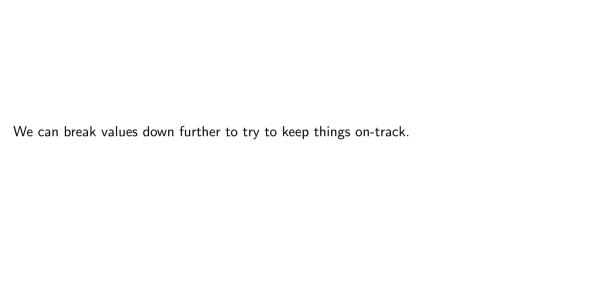
(E-IF)

This term is stuck:

iszero false

as is this one:

if O then false else true



# **Values**



nv





*bv* :=



false

true



succ nv







natural number value



boolean value



false value true value

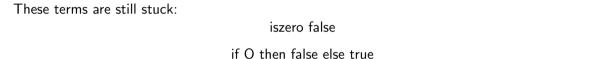


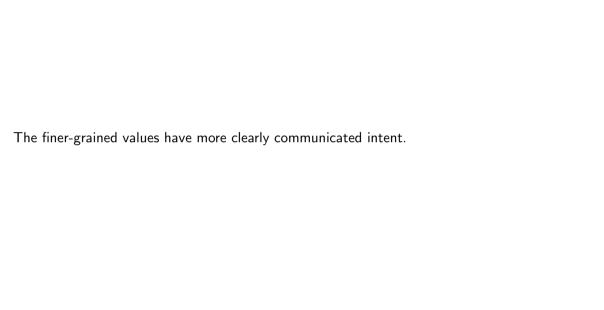


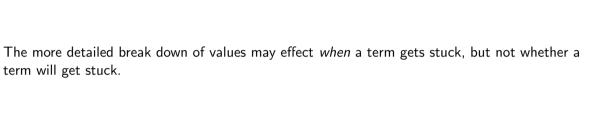
#### **Small-step semantics**

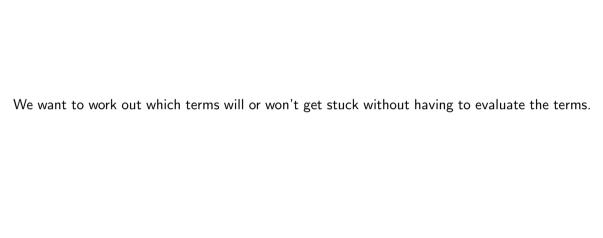
 $\overline{\mathsf{iszero}}$  (succ  $\overline{\mathit{nv}}$ )  $\longrightarrow$  false

(E-IsZeroSucc)











# **Types**

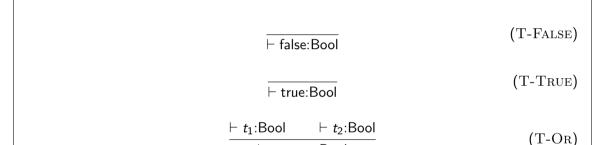
:=

Bool

Nat

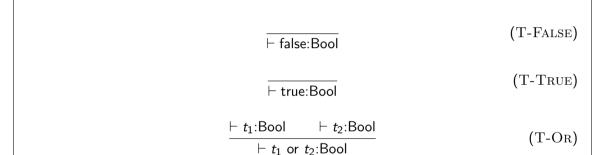
type of booleans

type of natural numbers

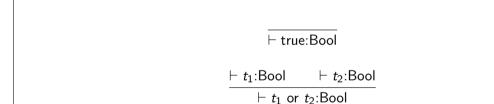


 $\vdash t_1 \text{ or } t_2:Bool$ 

We have a binary relation  $\vdash t:T$ .



This indicates that the term t is well-typed and has type T.



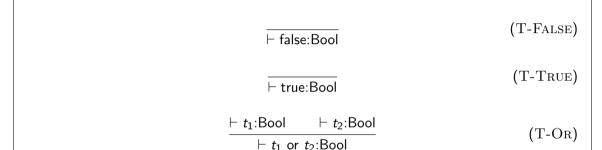
⊢ false:Bool

(T-False)

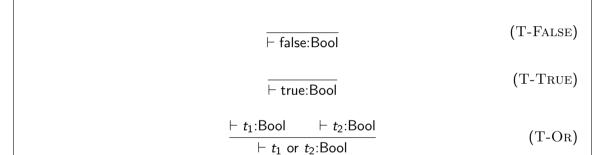
(T-True)

(T-OR)

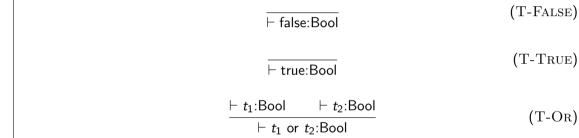
Any term which doesn't match any of these rules is *ill-typed*.



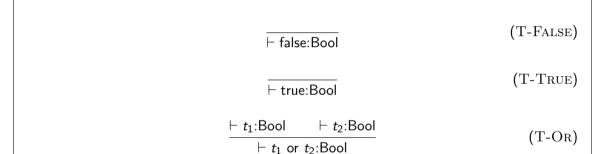
Anything on the left of the  $\vdash$  is additional context that the typing rules need.



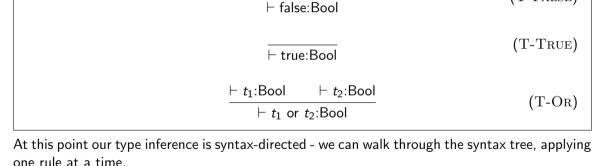
For now we don't need any more context.



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



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



(T-False)

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

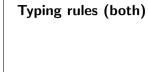
# Typing rules (both)

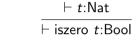
$$\frac{\vdash t:\mathsf{Nat}}{\vdash \mathsf{iszero}\ t:\mathsf{Bool}}$$

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

(T-IsZero)

(T-IF)





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

iszero false

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

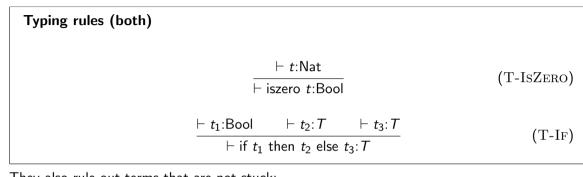
(T-IsZero)

(T-IF)

These rule out the stuck terms we saw previously:

and

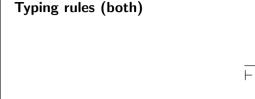
if O then false else true



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.



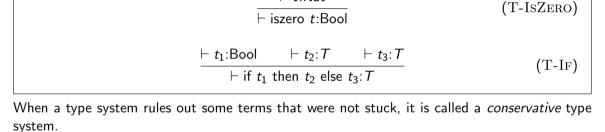
$$\frac{\vdash t:\mathsf{Nat}}{\vdash \mathsf{iszero}\ t:\mathsf{Bool}}$$

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

(T-IsZero)

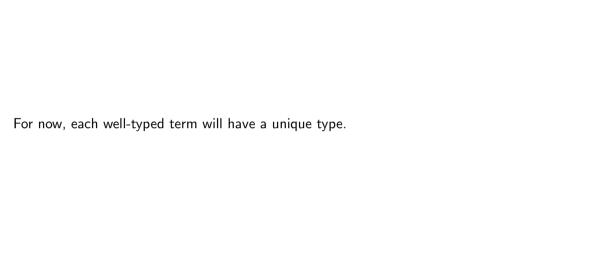
(T-IF)

This kind of thing is normally not a big deal.

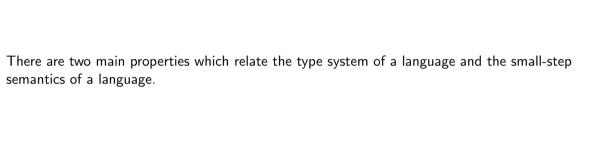


 $\vdash t:Nat$ 

Typing rules (both)



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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

if the term can take a step the type is unchanged .

For all well-typed terms,

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

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

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

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

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.

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.	

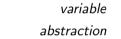
Together they mean we can	use type systems	to (approximately)	classify which	terms will or
will not get stuck.				





#### Terms and values

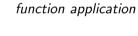
X  $\lambda x.t$ 



abstraction



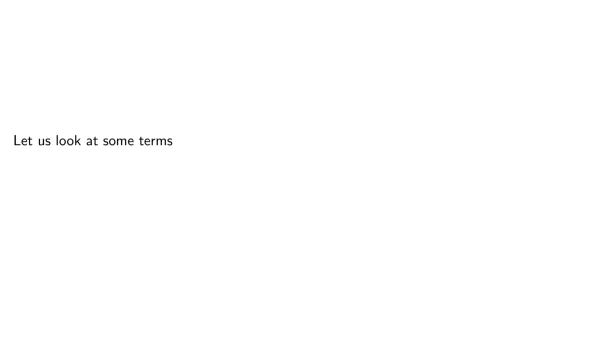








$$\lambda x.t$$



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

## Lambda anatomy

 $\lambda x \cdot x + 2$ 

### Lambda anatomy

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

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

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

f(1)

is equivalent to

when f is defined as before

In ordinary maths, we process

f(1) = 1 + 2

by taking

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

and replacing the occurrences of  $\boldsymbol{x}$  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:
$(\lambda \times \times + 2)1 \longrightarrow 1 + 2$

### Small-step semantics

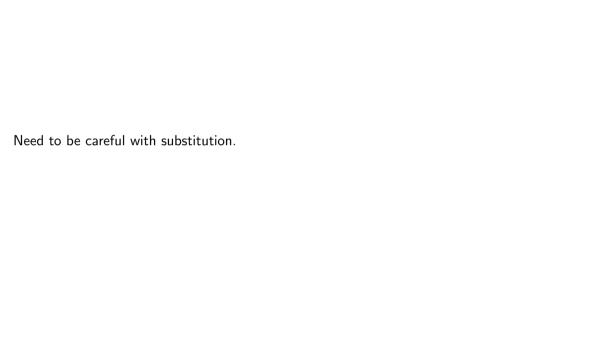
$$egin{aligned} rac{t_1 \longrightarrow t_1'}{t_1 \ t_2 \longrightarrow t_1' \ t_2} \ & \ rac{t_2 \longrightarrow t_2'}{v_1 \ t_2 \longrightarrow v_1 \ t_2'} \end{aligned}$$

 $(\lambda x.t_{12}) v_2 \longrightarrow [x \mapsto v_2] t_{12}$ 

(E-App2)

(E-APP1)

(E-APPLAM)



When we evaluate

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

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

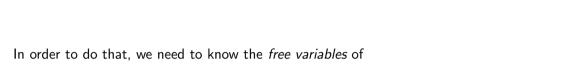
we have

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

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

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

to become



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



A variable is bound in a term if it appears inside a lambda abstraction with a matching variable

If there is no such lambda abstraction, then the variable is free in the term it appears in. |x| + 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

= ?

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)
```

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

```
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)
```

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

```
Free variable rules

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

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

```
Free variable rules

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

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

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

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

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

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

$$(\lambda x.t_1) = FV$$

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

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

$$= \emptyset$$

$$= FV$$

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

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

 $= \{x\} \cup \emptyset$ 

$$=\emptyset$$

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

$$|t\rangle$$
)  $=\emptyset$ 

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)
```

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

 $= \{x\}$ 

FV(x+1)

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

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

```
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)
```

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

```
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)
```

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

 $= \{x\}$ 

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)
```

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

 $= \{x\}$ 

FV(x+1)

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

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

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

```
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)
```

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

FV(x+1)

 $=\{x\}$ 

### Substitution rules

 $[x \mapsto s] x$ 

 $[x \mapsto s] v$ 

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

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

$$= s$$
  
 $= v$ 

$$= y$$

$$=\lambda v$$

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

= ([x 
$$\mapsto$$

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

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

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

$$[x \mapsto s] \iota_1)$$

$$t_1$$
) ([x

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

$$y \neq x$$

$$y \neq x$$

if 
$$y \neq x$$

$$y \neq x$$

$$y \neq x$$

 $[x \mapsto s] v$ 

$$[x \mapsto s] x$$

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

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

$$= s$$
  
 $= y$ 

$$= y$$

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

$$= \langle [x \mapsto s] t_1 \rangle$$

$$= ([$$

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

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

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

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

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

$$|t_1\rangle$$
 ([ $x\mapsto$ 

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

$$y \neq x$$

if 
$$y \neq x$$

$$y \neq x$$

$$\neq x$$

 $[x \mapsto s] v$ 

$$[x \mapsto s] x$$

$$= s$$
  
 $= v$ 

$$= y$$

$$=\lambda y$$

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

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

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

$$= ([x \mapsto 3] \iota_1)$$

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

$$=$$

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

$$\begin{array}{ccc} t_1 & t_2 \end{array} = ([$$

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

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

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

$$y \neq$$

if 
$$y \neq x$$

$$[x \mapsto s] x$$

$$= v$$

$$[x \mapsto s] y$$

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

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

$$(y.t_1) = \lambda$$

$$(y.t_1)$$
 =  $\lambda$ 

$$t_1, t_2$$
  $t_3$ 

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

$$t_1 t_2$$
 = ([

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

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

$$([x \mapsto s] \iota_1)$$

$$\kappa\mapsto s]\ t_1)$$

$$s] t_1) ([$$

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

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

if 
$$y \neq x$$

$$v \neq 1$$

 $[x \mapsto s] v$ 

$$[x \mapsto s] x$$

$$= v$$

$$= y$$

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

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

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

$$\mapsto s[t_1]$$

$$s] t_1)$$

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

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

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

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

if 
$$y \neq x$$
  
if  $y \neq x$ 

$$[x \mapsto s] x$$

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

$$=$$

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

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

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

$$(t_1 t_2)$$

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

$$t_1 t_2$$
 = ([x

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

$$= ([x \mapsto s] \iota_1,$$

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

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

$$ightarrow s] t_1)$$

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

$$t_1$$
)

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

$$f y \neq 0$$

$$y \neq x$$

if 
$$y \neq x$$

$$y \neq x$$

$$y \neq x$$

$$y \neq x$$

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

 $[x \mapsto s] v$ 

$$= y$$

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

$$(x, t_1) = 0$$

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

$$=([x\mapsto s]\iota_1)$$

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

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

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

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

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

$$y \neq$$

if 
$$y \neq x$$

$$y \neq y$$

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

$$= s$$
  
 $= y$ 

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

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

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

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

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

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

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

$$= \langle int \rangle$$

$$= \langle (x \mapsto s] t_1 \rangle + \langle (x \mapsto s] t_2 \rangle + \langle (x \mapsto s] t_3 \rangle + \langle (x \mapsto s] t_4 \rangle +$$

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

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

if 
$$y \neq x$$

 $[x \mapsto s] y$ 

$$[x \mapsto s] x$$

$$= v$$

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

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

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

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

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

$$[x \mapsto s] t_2$$

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

if 
$$y = 7$$

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

$$\neq x$$

 $[x \mapsto s] y$ 

$$[x \mapsto s] x$$

$$= y$$

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

$$(y,t_1) = \lambda$$

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

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

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

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

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

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

$$\mapsto s | t_2)$$

$$y \neq x$$

if 
$$y \neq x$$

$$y \neq x$$

$$y \neq x$$

$$v \neq x$$

$$v \neq x$$

 $[x \mapsto s] v$ 

$$[x \mapsto s] x$$

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

$$= s$$
  
 $= v$ 

$$= y$$

$$=\lambda$$

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

$$= \lambda y \cdot (|$$

$$=([x\mapsto$$

$$=([x\mapsto$$

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

$$(2) = (2)$$

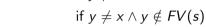
$$=([x$$

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

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

$$\mapsto s | t_1$$

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







 $[x \mapsto s] v$ 

$$[x \mapsto s] x$$

$$= s$$

$$= y$$

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

$$= \lambda y$$
.

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

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

$$t_2) = ([x \vdash$$

$$t_2$$
)

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

$$t_1 t_2) = ([x_1, t_2)]$$

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

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

$$s] t_1)$$

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

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

$$y \neq x$$

$$y \neq x$$

$$y \neq x$$

if 
$$y \neq x$$

$$\neq x$$

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

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

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

$$= y$$

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

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

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

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

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

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

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

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

if 
$$y \neq x$$

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

 $[x \mapsto s] v$ 

$$= y$$

$$= y$$
  
 $= \lambda$ 

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

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

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

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

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

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

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

$$[s] t_2)$$

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

$$y \neq x$$

if 
$$y \neq x$$





 $[x \mapsto s] v$ 

$$[x \mapsto s]x$$

$$= s$$
  
 $= y$ 

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

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

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

$$-([x\mapsto s]\iota_1)$$

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

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

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

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

$$\mapsto s] t_2)$$

$$y \neq x$$

if 
$$y \neq x$$

 $[x \mapsto s] v$ 

$$[x \mapsto s] x$$

$$= s$$
  
 $= y$ 

$$x y$$
  
 $x \lambda y \cdot ([x \mapsto s])$ 

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

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

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

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

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

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

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

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

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

if 
$$y \neq x$$

$$\neq x$$

 $[x \mapsto s] v$ 

$$[x \mapsto s] x$$

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

$$= v$$

= s

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

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

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

$$(t_1 \ t_2) = (t_2)$$

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

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

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

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

$$ightarrow s$$
]  $t_2$ )

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

$$y \neq x$$

$$y \neq x$$

if 
$$y \neq x$$

$$y \neq x$$

$$y \neq x$$

 $[x \mapsto s] v$ 

$$[x \mapsto s] x$$

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

$$= v$$

$$=\lambda$$

$$=\lambda$$

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

$$([x \mapsto :$$

$$=([x\mapsto$$

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

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

$$=([x \vdash$$

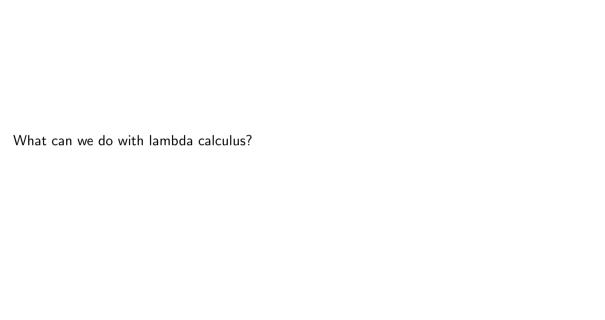
$$[x\mapsto s](t_1\ t_2) = ([x\mapsto s]\ t_1)\ ([x\mapsto s]\ t_2)$$

$$s \mid t_1)$$

$$([ \lor \hookrightarrow c]$$

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

if 
$$y \neq x$$



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)

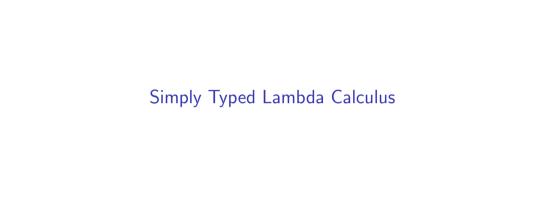
$$factorial = fix g$$

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



#### 

 $T \rightarrow T$ 

function arrow

We need some extra information to make the typing rules work.

#### 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.

## $(\lambda \times : T . t_1)t_2 \longrightarrow [x \mapsto t_2] t_1$

Unsurprisingly, the small-step semantics don't change.

**Small-step semantics** 

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

(E-APPLAM)

(E-App1)

(E-APP2)

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

Typing rules

$$\frac{x:T\in\Gamma}{\Gamma\vdash x:T}$$

 $\Gamma \vdash t_1 \ t_2 : T_2$ 

 $\Gamma \vdash t_1:T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2:T_1$ 

$$\frac{r_1 \rightarrow r_2}{\Gamma \vdash t_1 \ t_2 : T_2}$$

$$\Gamma, x : T_1 \vdash t : T_2$$

(T-VAR)

(T-App)

(T-ABS)

Now we need a context for our typing rules.

# $\frac{x:T \in \Gamma}{\Gamma \vdash x:T} \tag{T-VAR}$ $\frac{\Gamma \vdash t_1:T_1 \to T_2 \qquad \Gamma \vdash t_2:T_1}{\Gamma \vdash t_1 \ t_2:T_2} \tag{T-App}$ $\frac{\Gamma, x:T_1 \vdash t:T_2}{\Gamma \vdash (\lambda \ x:T_1.t):T_1 \to T_2} \tag{T-Abs}$

We use  $\Gamma$  as the context, which is a map from variables to types.

Typing rules

## $\Gamma \vdash (\lambda \times : T_1.t) : T_1 \rightarrow T_2$ T-Var just grabs the type from the context.

Typing rules

 $x:T \in \Gamma$ 

 $\Gamma \vdash t_1:T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2:T_1$  $\Gamma \vdash t_1 \ t_2 : T_2$ 

 $\Gamma, x: T_1 \vdash t: T_2$ 

(T-ABS)

(T-VAR)

(T-App)

#### $\Gamma \vdash t_1 \ t_2 : T_2$ $\Gamma, x: T_1 \vdash t: T_2$ $\Gamma \vdash (\lambda \times : T_1.t) : T_1 \rightarrow T_2$ A type error occurs if the variable isn't found in the context.

 $x:T \in \Gamma$ 

Γ ⊢ x: T

 $\Gamma \vdash t_1:T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2:T_1$ 

Typing rules

(T-VAR) (T-App)

(T-ABS)

Typing rules

 $\frac{x:T \in \Gamma}{\Gamma \vdash x:T}$  $\Gamma \vdash t_1:T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2:T_1$ 

(T-VAR) (T-App) $\Gamma \vdash t_1 \ t_2 : T_2$ 

 $\Gamma, x: T_1 \vdash t: T_2$  $\Gamma \vdash (\lambda x:T_1.t):T_1 \rightarrow T_2$  (T-ABS)

T-App has no new techniques in it.

 $\frac{\Gamma \vdash t_1 \ t_2 \colon I_2}{\Gamma, x \colon T_1 \vdash t \colon T_2} \\
\frac{\Gamma \vdash (\lambda \ x \colon T_1, t) \colon T_1 \to T_2}{\Gamma \vdash (\lambda \ x \colon T_1, t) \colon T_1 \to T_2}$ (T-Abs)

(T-VAR)

(T-App)

In T-Abs we temporarily add  $x:T_1$  to the context, just for long enough to find the type of t.

 $x:T \in \Gamma$ 

Typing rules

$$\frac{x:T \in \Gamma}{\Gamma \vdash x:T}$$

$$\Gamma \vdash t_1:T_1 \to T_2 \qquad \Gamma \vdash t_2:T_1$$

 $\frac{\Gamma \vdash t_2: T_1}{T_2} \tag{T-App}$ 

(T-VAR)

$$egin{array}{c} \Gamma ‐ t_1 \ t_2 \colon T_2 \ \hline \Gamma, x \colon T_1 ‐ t \colon T_2 \ \hline \Gamma ‐ (\lambda \ x \colon T_1 . t) \colon T_1 
ightarrow T_2 \end{array}$$

 $\frac{t:T_2}{:T_1 \to T_2} \tag{T-Abs}$ 

If we didn't modify the context then we would risk a type error occurring if the variable x appeared within the term t.

#### (T-App) $\Gamma \vdash t_1 \ t_2 : T_2$ $\Gamma, x: T_1 \vdash t: T_2$ $\Gamma \vdash (\lambda x: T_1.t): T_1 \rightarrow T_2$

 $x:T \in \Gamma$ 

 $\Gamma \vdash t_1: T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2: T_1$ 

(T-VAR)

(T-ABS)

Typing rules

With that done we know the type of the argument and of the result, so we are done.

$$\frac{}{\Gamma \vdash x : T}$$

$$\Gamma \vdash t_1 : T_1 \to T_2$$

$$\Gamma \vdash (\lambda \times : T_1 \cdot t) : T_1 \rightarrow T_2$$

$$\frac{\Gamma, x: T_1 \vdash t: T_2}{(\lambda x: T_1. t): T_1 \rightarrow T_2}$$

(T-VAR)

(T-App)

(T-ABS)

 $x:T\in\Gamma$ 

$$\begin{array}{cccc} \Gamma \vdash x : T \\ \hline \Gamma \vdash t_1 : T_1 \rightarrow T_2 & \Gamma \vdash t_2 : T_1 \end{array}$$

 $x : \mathsf{Bool} \vdash x : \mathsf{Bool}$ 

 $\Gamma \vdash t_1 \quad t_2 : T_2$ 

(T-VAR)

(T-App)

(T-ABS)

$$x : \mathsf{Bool} \in x : \mathsf{Bool}$$
 T-Var

 $x:T\in\Gamma$ 

 $\Gamma$ ,  $x: T_1 \vdash t: T_2$ 

 $\Gamma \vdash (\lambda \times : T_1 \cdot t) : T_1 \rightarrow T_2$ 

$$\begin{array}{c}
x : Bool \vdash x : Bool \\
\vdash \lambda x : Bool \cdot x : Bool \rightarrow Bool
\end{array}$$

$$\begin{array}{c}
T - Abs - Bool \\
\vdash (\lambda x : Bool \cdot x) \text{ true : Bool}
\end{array}$$

 $x : \mathsf{Bool} \vdash x : \mathsf{Bool}$ 

 $x:T\in\Gamma$ 

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

$$\frac{\Gamma, x: T_1 \vdash t: T_2}{\Gamma \vdash (\lambda \ x: T_1 \cdot t): T_1 \rightarrow T_2}$$

$$T_2$$

(T-VAR)

(T-App)

(T-ABS)

$$\frac{x : \mathsf{Bool} \in x : \mathsf{Bool}}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}} \mathsf{T}\mathsf{-}\mathsf{Var}$$

$$\frac{\mathsf{T}\mathsf{-}\mathsf{Abs}}{\vdash \lambda x : \mathsf{Bool} \cdot x : \mathsf{Bool} \rightarrow \mathsf{Bool}} \mathsf{T}\mathsf{-}\mathsf{Abs} \xrightarrow{\vdash \mathsf{true} : \mathsf{Bool}} \mathsf{T}\mathsf{-}\mathsf{True}$$

$$x : \mathsf{Bool} \to \mathsf{Bool} \to \mathsf{Fool} \to$$

# Typing rules $\frac{x: T \in \Gamma}{\Gamma \vdash x: T} \tag{T-VAR}$ $\frac{\Gamma \vdash t_1: T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2: T_1}{\Gamma \vdash t_1 \quad t_2: T_2} \tag{T-APP}$ $\frac{\Gamma, x: T_1 \vdash t: T_2}{\Gamma \vdash (\lambda \ x: T_1 \ t): T_1 \rightarrow T_2} \tag{T-ABS}$

$$\frac{x : \mathsf{Bool} \in x : \mathsf{Bool}}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}} \mathsf{T\text{-}Var}$$

$$\frac{\mathsf{T\text{-}True}}{\mathsf{L} \times \mathsf{Sool} \times \mathsf{Sool} \times \mathsf{Bool}} \mathsf{T\text{-}Abs} \xrightarrow{\mathsf{L} \times \mathsf{Sool}} \mathsf{T\text{-}App}$$

$$\mathsf{L} \times \mathsf{Sool} \times \mathsf$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash t_1 : T_1 \rightarrow T_2} \qquad \Gamma \vdash t_1 \\
\Gamma \vdash t_1 \qquad t_2 : T_2 \\
\Gamma, x : T_1 \vdash t : T_2$$

 $x:T\in\Gamma$ 

 $\Gamma \vdash (\lambda \times : T_1 \cdot t) : T_1 \rightarrow T_2$ 

$$\frac{x : \mathsf{Bool} \in x : \mathsf{Bool}}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}} \mathsf{T-Var}$$

$$\vdash \lambda x : \mathsf{Bool} \cdot x : \mathsf{Bool} \to \mathsf{Bool} \mathsf{T-Abs} \xrightarrow{\vdash \mathsf{true} : \mathsf{Bool}} \mathsf{T-App}$$

(T-VAR)

(T-App)

(T-ABS)

(T-VAR)

(T-App)

(T-ABS)

$$\frac{x : \mathsf{Bool} \in x : \mathsf{Bool}}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}} \mathsf{T-Var} \\ \vdash \lambda x : \mathsf{Bool} \cdot x : \mathsf{Bool} \to \mathsf{Bool} \\ \vdash (\lambda x : \mathsf{Bool} \cdot x) \mathsf{true} : \mathsf{Bool}$$
$$\mathsf{T-App}$$

 $x:T\in\Gamma$ 

 $\Gamma$ ,  $x: T_1 \vdash t: T_2$ 

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (\lambda \ x : T_1 . t \ ) : T_1 \rightarrow T_2}$$

$$\frac{x : \mathsf{Bool} \in x : \mathsf{Bool}}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}} \mathsf{T-Var}$$

$$\vdash \lambda x : \mathsf{Bool} . x : \mathsf{Bool} \rightarrow \mathsf{Bool}$$

$$\vdash (\lambda x : \mathsf{Bool} x) \mathsf{true} : \mathsf{Bool}$$

$$\mathsf{T-App}$$

(T-VAR)

(T-App)

(T-ABS)

 $x:T\in\Gamma$ 

 $\Gamma \vdash x : T$ 

#### 

$$\frac{x : \mathsf{Bool} \in x : \mathsf{Bool}}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}} \mathsf{T-Var}$$

$$\frac{\mathsf{T-Tr}}{\mathsf{T-Als}} = \frac{\mathsf{T-Tr}}{\mathsf{T-Als}} = \frac{\mathsf{T-Tr}}{\mathsf{T-Als}} = \frac{\mathsf{T-Tr}}{\mathsf{T-Als}}$$

#### 

$$\frac{x : \mathsf{Bool} \in x : \mathsf{Bool}}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}} \mathsf{T-Var}$$

$$\frac{\mathsf{T-Tr}}{\mathsf{Fool} \times \mathsf{Bool} \times \mathsf{Bool}} \mathsf{T-Ap}$$

$$\mathsf{Fool} \times \mathsf{Bool} \times \mathsf{Bool} \times \mathsf{Bool}$$

$$\mathsf{T-Ap}$$

$$\overline{\Gamma \vdash x : T}$$

$$\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1$$

(T-VAR)

(T-App)

(T-ABS)

$$\mathsf{Bool} \in x : \mathsf{Bool} \ \_ \dots$$

$$\frac{x : \mathsf{Bool} \in x : \mathsf{Bool}}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}} \mathsf{T-Var}$$

$$\frac{\mathsf{T-True}}{\mathsf{T-Abs}} \xrightarrow{\mathsf{T-True} : \mathsf{Bool}} \mathsf{T-App}$$

$$\mathsf{T-App}$$

$$\mathsf{T-App}$$

 $x:T\in\Gamma$ 

 $\Gamma$ ,  $x: T_1 \vdash t: T_2$ 

 $\Gamma \vdash (\lambda \times : T_1 \cdot t) : T_1 \rightarrow T_2$ 

$$\frac{}{\Gamma \vdash x : T}$$

$$\Gamma \vdash t_1 : T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2 : T_1$$

$$T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2 : T_1 \ \Gamma \vdash t_1 \quad t_2 : T_2$$
 (T-App)

(T-VAR)

(T-ABS)

$$x : \mathsf{Bool} \in x : \mathsf{Bool}$$
 T-Var

 $x:T\in\Gamma$ 

 $\Gamma$ ,  $x: T_1 \vdash t: T_2$ 

 $\Gamma \vdash (\lambda \times : T_1 \cdot t) : T_1 \rightarrow T_2$ 

# Typing rules $\frac{x: T \in \Gamma}{\Gamma \vdash x: T} \tag{T-VAR}$ $\frac{\Gamma \vdash t_1: T_1 \to T_2 \qquad \Gamma \vdash t_2: T_1}{\Gamma \vdash t_1 \quad t_2: T_2} \tag{T-App}$ $\frac{\Gamma, x: T_1 \vdash t: T_2}{\Gamma \vdash (\lambda \times : T_1 - t): T_1 \to T_2} \tag{T-Abs}$

$$\frac{x : \mathsf{Bool} \in x : \mathsf{Bool}}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}} \mathsf{T-Var}$$

$$\frac{\mathsf{T-True}}{\vdash \lambda x : \mathsf{Bool} \cdot x : \mathsf{Bool}} \to \mathsf{Bool} \to \mathsf{Bool}$$

$$\vdash (\lambda x : \mathsf{Bool} \cdot x) \mathsf{true} : \mathsf{Bool} \to \mathsf{Bool}$$

## Typing rules $\frac{x: T \in \Gamma}{\Gamma \vdash x: T} \tag{T-VAR}$ $\frac{\Gamma \vdash t_1: T_1 \to T_2 \qquad \Gamma \vdash t_2: T_1}{\Gamma \vdash t_1 \quad t_2: T_2} \tag{T-APP}$ $\frac{\Gamma \vdash t_1 \quad t_2: T_2}{\Gamma \vdash t_1 \quad t_2: T_2} \tag{T-ABS}$

$$\frac{x : \mathsf{Bool} \in x : \mathsf{Bool}}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}} \mathsf{T-Var}$$

$$\frac{\mathsf{Ax} : \mathsf{Bool} \vdash x : \mathsf{Bool}}{\vdash \lambda \mathsf{x} : \mathsf{Bool} \cdot \mathsf{x} : \mathsf{Bool}} \mathsf{T-Abs} \xrightarrow{\vdash \mathsf{true} : \mathsf{Bool}} \mathsf{T-App}$$

$$\vdash (\lambda x : \mathsf{Bool}.x) \mathsf{true} : \mathsf{Bool}$$

## Typing rules $\frac{x: T \in \Gamma}{\Gamma \vdash x: T} \tag{T-VAR}$ $\frac{\Gamma \vdash t_1: T_1 \to T_2 \qquad \Gamma \vdash t_2: T_1}{\Gamma \vdash t_1 \quad t_2: T_2} \tag{T-APP}$ $\frac{\Gamma \vdash (\lambda \times : T_1 \vdash t: T_2)}{\Gamma \vdash (\lambda \times : T_1 \cdot t): T_1 \to T_2} \tag{T-ABS}$

$$\frac{x : \mathsf{Bool} \in x : \mathsf{Bool}}{|\mathsf{x}| : \mathsf{Bool} \mid \mathsf{x}| : \mathsf{Bool}} \mathsf{T-Var}$$

$$\frac{|\mathsf{bool}| \cdot |\mathsf{x}| : \mathsf{Bool}|}{|\mathsf{bool}| \cdot |\mathsf{x}| : \mathsf{Bool}} \mathsf{T-Abs} \xrightarrow{\mathsf{bool}} \mathsf{T-App}$$

$$\frac{|\mathsf{bool}| \cdot |\mathsf{x}| : \mathsf{Bool}| \cdot |\mathsf{x}| : \mathsf{Bool}|}{|\mathsf{bool}| \cdot |\mathsf{x}| : \mathsf{Bool}|} \mathsf{T-App}$$

 $\frac{\Gamma \vdash t_1 \quad t_2 : T_2}{\Gamma , \ x : T_1 \vdash t : T_2}$   $\frac{\Gamma \vdash (\lambda \quad x : T_1 \cdot t \ ) : T_1 \rightarrow T_2}{\Gamma \vdash (\lambda \quad x : T_1 \cdot t \ ) : T_1 \rightarrow T_2}$ 

(T-VAR)

(T-App)

(T-ABS)

 $x:T\in\Gamma$ 

$$\frac{}{\Gamma \vdash x : T}$$

$$\Gamma \vdash t_1 : T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2 : T_1$$

 $\vdash \lambda \ x : \mathsf{Bool} \ . \ x : \mathsf{Bool} \ o \ \mathsf{Bool}$ 

 $x:T\in\Gamma$ 

 $\Gamma$ ,  $x: T_1 \vdash t: T_2$ 

 $\Gamma \vdash t_1 \quad t_2 : T_2$ 

(T-VAR)

(T-App)

(T-ABS)

$$\Gamma \vdash (\lambda \times : T_1 \cdot t) : T_1 \rightarrow T_2$$

# Typing rules $\begin{array}{c} x: T \in \Gamma \\ \hline{\Gamma \vdash x}: T \end{array} \qquad (T-VAR)$ $\frac{\Gamma \vdash t_1: T_1 \to T_2 \qquad \Gamma \vdash t_2: T_1}{\Gamma \vdash t_1 \quad t_2: T_2} \qquad (T-App)$ $\frac{\Gamma, x: T_1 \vdash t: T_2}{\Gamma \vdash (\lambda \quad x: T_1 \cdot t): T_1 \to T_2} \qquad (T-App)$

$$\frac{x : \mathsf{Bool} \in x : \mathsf{Bool}}{\mathsf{x:Bool} \vdash \mathsf{x} : \mathsf{Bool}} \mathsf{T-Var} \\ \frac{\mathsf{T-True}}{\mathsf{E} \mathsf{A} \mathsf{x} : \mathsf{Bool} \cdot \mathsf{x} : \mathsf{Bool} \to \mathsf{Bool}} \mathsf{T-Abs} \\ \mathsf{E} \mathsf{A} \mathsf{x} : \mathsf{Bool} \cdot \mathsf{x} : \mathsf{Bool} \to \mathsf{Bool}} \mathsf{T-App}$$

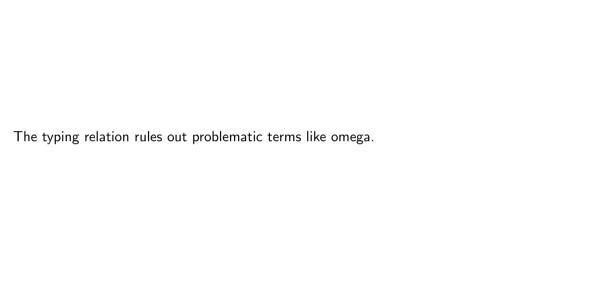
#### 

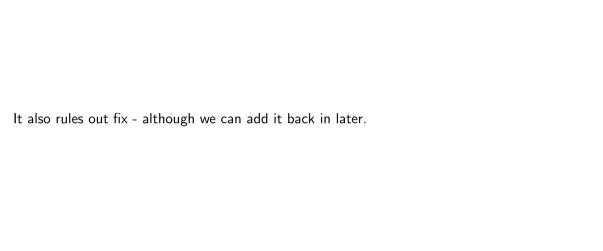
$$\frac{x : \mathsf{Bool} \in x : \mathsf{Bool}}{x : \mathsf{Bool}} \mathsf{T-Var}$$

$$\frac{\mathsf{T-True}}{\mathsf{E} \times \mathsf{Bool} \times \mathsf{Bool} \times \mathsf{Bool}} \mathsf{T-Abs} \xrightarrow{\mathsf{E} \times \mathsf{Bool}} \mathsf{T-App}$$

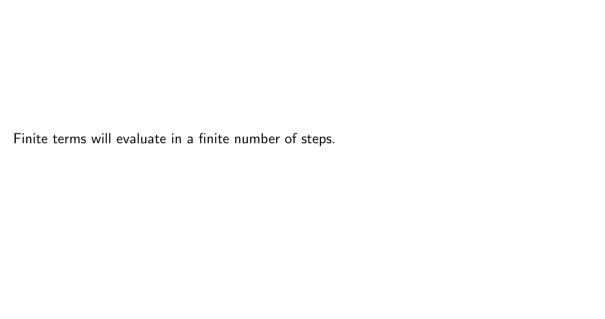
$$\mathsf{E} \times \mathsf{E} \times \mathsf{Bool} \times \mathsf{E} \times \mathsf{Bool} \times \mathsf{E} \times \mathsf{Bool} \times \mathsf{E} \times \mathsf{E}$$

# Typing rules $\begin{array}{c} x: T \in \Gamma \\ \hline \Gamma \vdash x: T \end{array}$ $\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 \quad t_2 : T_2} \qquad (T-App)$ $\frac{\Gamma, x: T_1 \vdash t: T_2}{\Gamma \vdash (\lambda \ x: T_1 . t ): T_1 \rightarrow T_2} \qquad (T-Abs)$





t rules out enough problematic terms that STLC is actually strongly normalizing.	





#### Terms

$$t := \ldots$$

fix t fixed point

### Small-step semantics

$$\dfrac{t\longrightarrow t'}{\mathit{fix}\ t\longrightarrow \mathit{fix}\ t'}$$

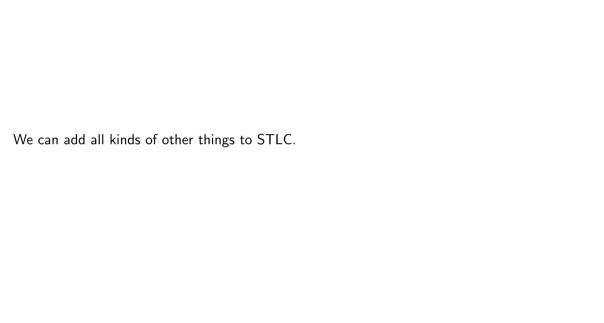
 $\overline{\text{fix}(\lambda x:T.t)} \longrightarrow [x \mapsto \text{fix}(\lambda x:T.t)] t$ 

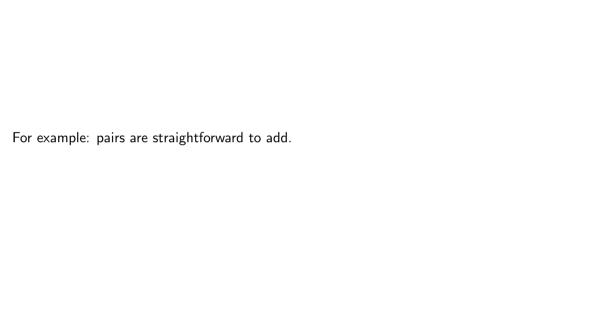
(E-Fix1)

(E-FIXBETA)

$$\frac{\vdash t: T \to T}{\vdash \textit{fix } t: T}$$

(T-Fix)





### 

(v, v)

 $T \times T$ 

pair introduction

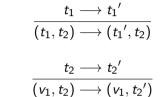
pair elimination

pair elimination

pair value

pair type

## **Small-step semantics**



(E-Pair2)

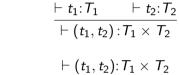
(E-Pair1)

(E-SNDPAIR)

$$fst(v_1, v_2) \longrightarrow v_1$$

 $snd(v_1, v_2) \longrightarrow v_2$ 

$$\longrightarrow (v_1,t_2{}')$$



 $\vdash$  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)

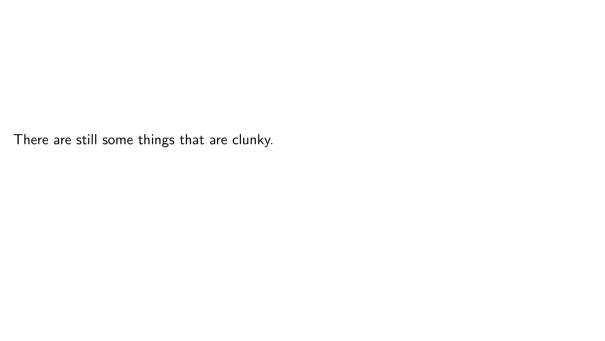
(T-Pair)

(T-PairSnd)



Tuples, records, variants and lists are similar.

Recursive types require a bit more work - and we'd need to get into pattern we don't have time for.	matching,	which



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 .  $x$ 

$$\lambda x : \mathsf{Bool} \cdot \lambda y : \mathsf{Int} \cdot x$$
  
 $\lambda x : \mathsf{Int} \cdot \lambda y : \mathsf{Bool} \cdot x$ 

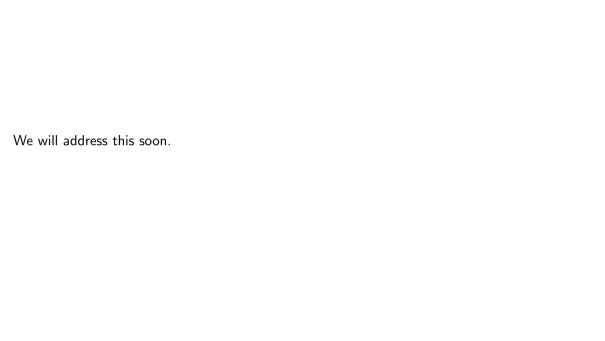
$$\lambda x : \mathsf{Int} \cdot \lambda y : \mathsf{Bool} \cdot x$$

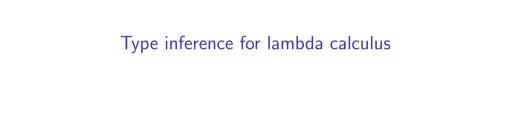
 $\lambda x : Int . \lambda y : Int . x$ 

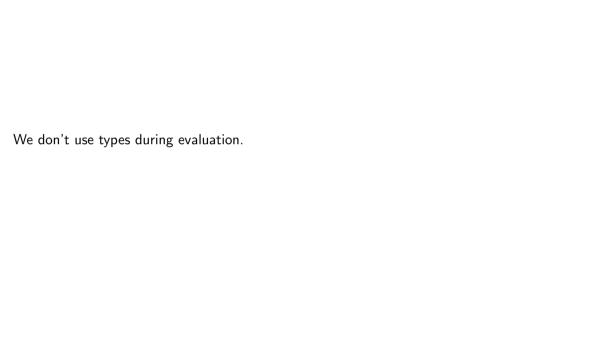
Don't even get me started on compose

$$\lambda \ f : \mathsf{Bool} \to \mathsf{Int} \ . \ \lambda \ g : \mathsf{Int} \to \mathsf{Bool} \ . \ \lambda \ x : \mathsf{Int} \ . \ f \ (g \ x)$$

. . .



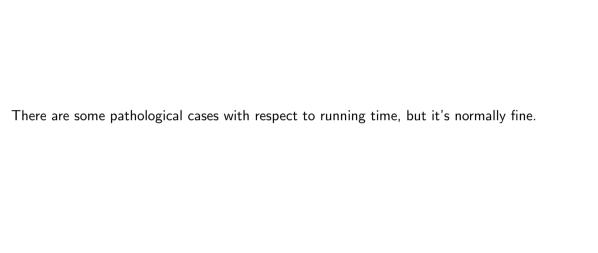




So we can check that a type is well-typed and then *erase* the type annotations, and the term should still evaluate.

The reverse idea - starting with an unannotated term and recovering the type information - type reconstruction.

The Hindley-Milner-Damas algorithm is used to do this for STLC.	

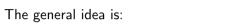


We no longer have enough information to do syntax-directed type inference - walking the
syntax tree and applying rules as we go.



The general idea is:

► Generate type variables all over the place



- ► Generate type variables all over the place

▶ Record constraints on the type variables when we run into something concrete

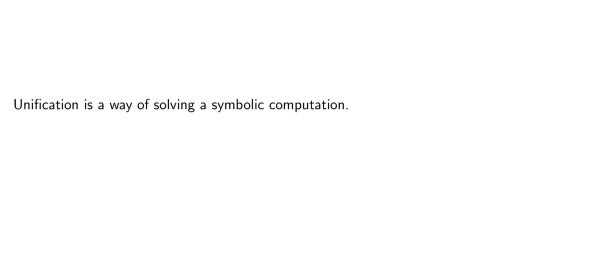
## The general idea is:

- ► Generate type variables all over the place
- ▶ Record constraints on the type variables when we run into something concrete
- Record constraints on the type variables when we run into something concrete

*Unify* these constraints to find a map from type variables to types

## The general idea is:

- ► Generate type variables all over the place
- ▶ Record constraints on the type variables when we run into something concrete
- ► *Unify* these constraints to find a map from type variables to types
- Use that map to replace all the type variables with types.



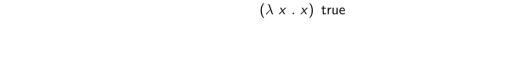
Unification is a way of solving a symbolic computation.

▶ When is  $[1, a, [2, 3], b] \sim [c, [3, 4], d, e]$ ?

Unification is a way of solving a symbolic computation.

▶ When we have  $\{1 \sim c, a \sim [3, 4], [2, 3] \sim d, b \sim e\}$ 

- ▶ When is  $[1, a, [2, 3], b] \sim [c, [3, 4], d, e]$ ?



Let us look at our old friend:



$$\lambda \times . t$$

we expect that given

$$\Gamma, x : C \vdash t : D$$

 $\lambda \times . t$ 

we expect that given

 $\Gamma, x : C \vdash t : D$ 

that the overall type will be

 $\lambda \times . \ t : C \rightarrow D$ 

 $\lambda \times . t$ 

we expect that given

 $\Gamma, x : C \vdash t : D$ 

 $\lambda \times t: C \to D$ 

Ø

that the overall type will be

with constraints

$$\lambda \times . \times$$

we expect that given

with constraints

$$\Gamma, x : C \vdash t : D$$

that the overall type will be

Ø

 $\lambda \times t: C \to D$ 

$$\lambda x . x$$

we expect that given

with constraints

$$\Gamma, x : C \vdash x : D$$

that the overall type will be

 $\lambda \times t: C \to D$ 

 $\lambda \times . \times$ 

 $\left\{ C \sim D \right\}$ 

we expect that given

 $\Gamma, x : C \vdash x : D$ 

that the overall type will be

 $\lambda \times t: C \to D$ 

with constraints

 $\lambda \times . \times$ 

we expect that given

 $\Gamma, x : C \vdash x : D$ 

that the overall type will be

 $\lambda \times . \times : C \rightarrow D$ 

with constraints

 $\left\{ \ C\sim D\ \right\}$ 

 $\lambda \times . \times$ 

we expect that given

 $\Gamma, x : C \vdash x : D$ 

that the overall type will be

with constraints

 $\lambda \times ... \times : C \rightarrow D$ 

 $\left\{ C \sim D \right\}$ 

 $\lambda \times . \times$ 

we expect that given

 $\lambda \times . \times : C \to D$ 

 $\{C \sim D\}$ 

with constraints

that the overall type will be

 $\Gamma, x : C \vdash x : D$ 

 $\lambda \times . \times$ 

we expect that given

 $\Gamma, x : C \vdash x : C$ 

that the overall type will be

 $\{C \sim D\}$ 

 $\lambda \times . \times : C \rightarrow C$ 

with constraints

$$\lambda \times . \times$$

we expect that given

with constraints

$$\Gamma, x : C \vdash x : C$$

that the overall type will be

$$\lambda \times . \times : C \to C$$

We know that

mat

⊢ true : Bool





we expect that given

 $\vdash f: A \rightarrow B$ 

and

⊢ x : A

f x

we expect that given

f x

 $\vdash f : A \rightarrow B$ 

 $\vdash x : A$ 

 $\vdash f x : B$ 

and

that the overall type will be

we expect that given

and

that the overall type will be

with constraints

 $\vdash f x : B$ 

f x

 $\vdash f : A \rightarrow B$ 

 $\vdash x : A$ 

Ø

and

 $(\lambda \times ... \times) \times$ we expect that given

 $\vdash f : A \rightarrow B$  $\vdash x : A$ 

that the overall type will be

with constraints

Ø

 $\vdash f x : B$ 

and

that the overall type will be

with constraints

 $\vdash f x : B$ 

 $(\lambda \times ... \times) \times$ 

 $\vdash (\lambda \times ... \times) : A \rightarrow B$ 

 $\vdash x : A$ 

Ø

and

we expect that given

that the overall type will be

with constraints

 $\vdash f \times : B$ 

 $(\lambda \times ... \times) \times$ 

 $\vdash (\lambda \times ... \times) : A \rightarrow B$ 

 $\vdash x : A$ 

 $\left\{\left(\begin{array}{ccc}A \rightarrow B\end{array}\right) \sim \left(\begin{array}{ccc}C \rightarrow C\end{array}\right)\right\}$ 

For

we expect that given

that the overall type will be

with constraints

 $\vdash f \times : B$ 

 $\left\{ A \sim C, B \sim C \right\}$ 

 $(\lambda \times ... \times) \times$ 

 $\vdash (\lambda \times ... \times) : A \rightarrow B$ 

 $\vdash x : A$ 

we expect that given

and

that the overall type will be

with constraints

 ${A \sim C, B \sim C}$ 

 $\vdash (\lambda \times ... \times) \times : B$ 

 $(\lambda \times ... \times) \times$ 

 $\vdash (\lambda \times ... \times) : A \rightarrow B$ 

 $\vdash x : A$ 

For

we expect that given

that the overall type will be

with constraints

 $\vdash (\lambda \times ... \times) \times : B$ 

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda \times ... \times) : A \rightarrow B$ 

 $\vdash x : A$ 

 ${A \sim C, B \sim C}$ 

For

we expect that given

that the overall type will be

with constraints

 $\vdash (\lambda \times ... \times) \times : B$ 

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda \times ... \times) : A \rightarrow B$ 

 $\vdash$  true : A

 ${A \sim C, B \sim C}$ 

and

we expect that given

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda \times ... \times) : A \rightarrow B$ 

 $\vdash$  true : A

 $\vdash (\lambda \times ... \times) \times : B$ 

 $\left\{ A \sim \text{Bool}, A \sim C, B \sim C \right\}$ 

that the overall type will be

with constraints

and

we expect that given

that the overall type will be

with constraints

 $\vdash$  true : A

 $\left\{ A \sim \text{Bool}, A \sim C, B \sim C \right\}$ 

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda \times ... \times) : A \rightarrow B$ 

 $\vdash (\lambda \times ... \times)$  true : B

we expect that given

and

that the overall type will be

with constraints

 $\vdash (\lambda \times ... \times)$  true : B

 $\vdash$  true : A

 $\left\{ A \sim \text{Bool}, A \sim C, B \sim C \right\}$ 

 $\vdash (\lambda \times ... \times) : A \rightarrow B$ 

 $(\lambda \times ... \times)$  true

For

we expect that given

that the overall type will be

with constraints

 $\left\{ A \sim C \sim \text{Bool}, B \sim C \right\}$ 

 $\vdash (\lambda \times ... \times)$  true : B

 $(\lambda \times ... \times)$  true

 $\vdash$  true : A

 $\vdash (\lambda \times ... \times) : A \rightarrow B$ 

For

ve	expect	that	given	

that the overall type will be

with constraints

 $\vdash (\lambda \times ... \times)$  true : B

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda \times ... \times) : A \rightarrow B$ 

 $\vdash$  true : A

 $\left\{ A \sim C \sim \text{Bool}, B \sim C \right\}$ 

For we expect that given

	(,,,,,	.,		
⊢ ( <i>λ</i>	x . x)	:	$A \rightarrow$	В

 $(\lambda \times x)$  true

with constraints

$$\vdash (\lambda \ x \ . \ x) \ \mathsf{true} : \ B$$
  $\Big\{ \ A \ \sim \ C \ \sim \mathsf{Bool}, \ B \ \sim \ C \Big\}$ 

that the overall type will be

For

we expect that given

that the overall type will be

with constraints

 $\vdash (\lambda \times ... \times)$  true : B

 ${A \sim B \sim C \sim Bool}$ 

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda \times ... \times) : A \rightarrow B$ 

 $\vdash$  true : A

For

we	expect	that	given

that the overall type will be

with constraints

 $\vdash (\lambda \times ... \times)$  true : B

 ${A \sim B \sim C \sim Bool}$ 

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda \times ... \times) : A \rightarrow B$ 

 $\vdash$  true : A

For

we expect that given

with constraints

that the overall type will be

 $\vdash (\lambda \times ... \times)$  true : B

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda \times ... \times) : A \rightarrow B$ 

 $\vdash$  true : A

For

we expect that given

that the overall type will be

with constraints

 $\vdash (\lambda \times ... \times)$  true : B

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda \times ... \times) : A \rightarrow B$ 

 $\vdash$  true : A

$$\{A \sim B \sim C \sim Bool\}$$

For

we expect that given

that the overall type will be

with constraints

 $\vdash$  true : A

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda \times ... \times) : Bool \rightarrow B$ 

 $\vdash (\lambda \times ... \times)$  true : B

For

we expect that given

that the overall type will be

with constraints

 $\vdash (\lambda \times ... \times)$  true : B

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda \times ... \times) : Bool \rightarrow B$ 

 $\vdash$  true : A

and

we expect that given

that the overall type will be

with constraints

 $\vdash (\lambda \times ... \times)$  true : B

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda \times ... \times) : \mathsf{Bool} \to B$ 

 $\vdash$  true : A

For

we expect that given

that the overall type will be

with constraints

 $\vdash (\lambda \times ... \times)$  true : B

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda x . x) : \mathsf{Bool} \to \mathsf{Bool}$ 

 $\vdash$  true : A

For

we expect that given

that the overall type will be

with constraints

 $\vdash (\lambda \times ... \times)$  true : B

 $\vdash$  true : A

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda x . x) : \mathsf{Bool} \to \mathsf{Bool}$ 

For

we expect that given

that the overall type will be

with constraints

 $\vdash (\lambda \times ... \times)$  true : B

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda x . x) : \mathsf{Bool} \to \mathsf{Bool}$ 

 $\vdash$  true : A

For

we expect that given

that the overall type will be

with constraints

 $\vdash (\lambda \times ... \times)$  true : B

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda x . x) : \mathsf{Bool} \to \mathsf{Bool}$ 

⊢ true : Bool

For

we expect that given

and

that the overall type will be

with constraints

 $\vdash (\lambda \times ... \times)$  true : B

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda x . x) : \mathsf{Bool} \to \mathsf{Bool}$ 

⊢ true : Bool

For

we	expect	that	given

that the overall type will be

with constraints

 $\vdash (\lambda \times ... \times)$  true : B

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda x . x) : \mathsf{Bool} \to \mathsf{Bool}$ 

⊢ true : Bool

For

we expect that given

that the overall type will be

with constraints

 $\vdash (\lambda \times ... \times)$  true : Bool

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda x . x) : \mathsf{Bool} \to \mathsf{Bool}$ 

⊢ true : Bool

and

we expect that given

 $(\lambda \times ... \times)$  true

 $\vdash (\lambda x . x) : \mathsf{Bool} \to \mathsf{Bool}$ 

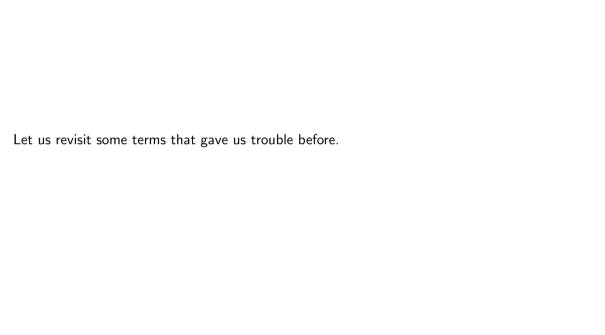
⊢ true : Bool

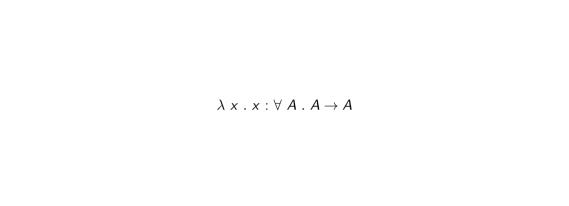
 $\vdash (\lambda \times ... \times)$  true : Bool

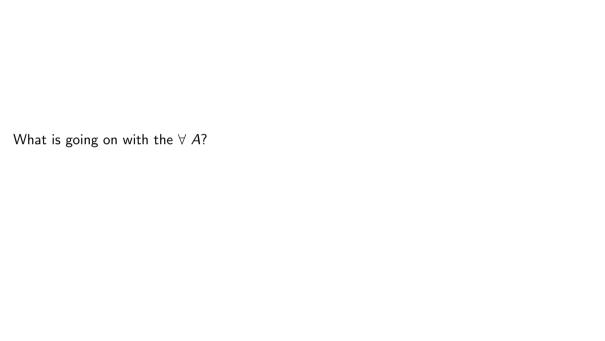
Ø

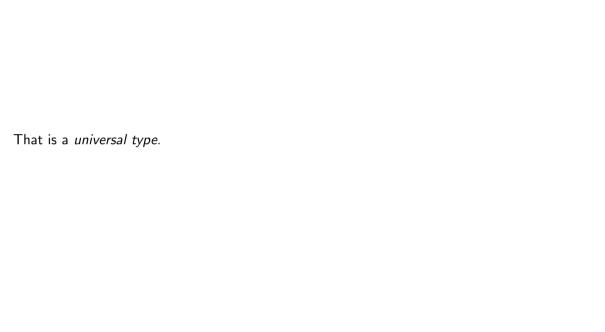
with constraints

that the overall type will be



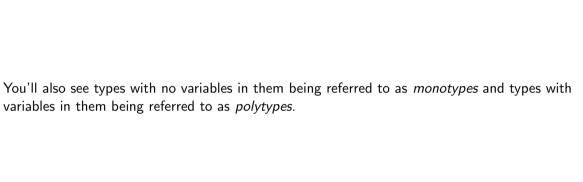






This is what we get when we our type-type-checking.	e still has unconstrained	type variables in it at the end

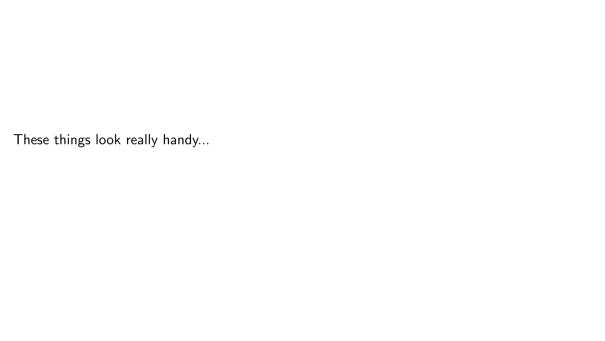
You'll see references to types that occur in the inference for a language but not in the language itself as a <i>type scheme</i> .



There universal types were handy for id - let us see how it fares with const and co	mpose



 $\lambda \ f \ . \ \lambda \ g \ . \ \lambda \ x \ . \ f \ (g \ x) \ : \forall \ A \ . \ \forall \ B \ . \ \forall \ C \ . \ (B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C)$ 





Universal types: System F

#### **Types**

 $\forall X.T$ 

type variable  $T \rightarrow T$  function arrow

univeral type





We start by adding universal types into the mix

#### variable Х $\lambda \times T.t$ abstraction function application t t $\lambda X.t$ type abstraction type application $\lambda \times T.t$ abstraction $\lambda X.t$ type abstraction

Terms and values

We need the ability to abstract over a type, and to later supply a type to that abstraction.

# Terms and values

We abstract over types to do things like id:

$$\lambda X \cdot \lambda x : X \cdot x : \forall A \cdot A \rightarrow A$$

### Terms and values

$$\lambda \ x:T.t \qquad abstraction$$

$$t \ t \qquad function \ application$$

$$\lambda \ X.t \qquad type \ abstraction$$

$$t \ [T] \qquad type \ application$$

$$v = \dots$$

$$\lambda \ x:T.t \qquad abstraction$$

$$\lambda \ X.t \qquad type \ abstraction$$

Х

variable

We supply concrete types later on to be able to use these terms:

$$(\lambda \ X \ . \ \lambda \ x : X \ . \ x)$$
 [Int] : Int  $\rightarrow$  Int

#### $\lambda X.t$ type abstraction t [T] type application $\lambda \times T.t$ abstraction $\lambda X.t$ type abstraction If we didn't have these things to guide the way, we'd lose syntax-directed type checking / inference

Х

t t

 $\lambda \times T.t$ 

variable

abstraction

function application

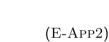
Terms and values

### Small-step semantics (old)

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

 $(\lambda x:T.t_1)t_2 \longrightarrow [x \mapsto t_2]t_1$ 

(E-App1)



(E-Appabs)

$$t_1$$
  $t_2$ 

$$\_\_\_ t_1 \longrightarrow t_1{'}$$

 $(\lambda X.t_{12})[T_2] \longrightarrow [X \mapsto T_2]t_{12}$ 

 $\frac{t_1 \longrightarrow t_1{}'}{t_1 \ [T_2] \longrightarrow t_1{}' \ [T_2]}$ 

(E-TAPP)

(E-TAPPTABS)

The new rules are straightforward.

 $\frac{t_1 \longrightarrow t_1'}{t_1 \ [T_2] \longrightarrow t_1' \ [T_2]}$ 

(E-TAPP)

$$t_1 \ [I_2] \longrightarrow t_1'$$

(E-TAPPTABS)

 $\overline{(\lambda X.t_{12})[T_2] \longrightarrow [X \mapsto T_2]t_{12}}$ 

E-TApp says we evaluate the terms inside type applications.

$$\frac{t_1 \longrightarrow t_1'}{t_1 \ [T_2] \longrightarrow t_1' \ [T_2]} \tag{E-TAPP}$$

$$\frac{(E-TAPP)}{(\lambda \ X.t_{12}) \ [T_2] \longrightarrow [X \mapsto T_2] \ t_{12}} \tag{E-TAPPTABS}$$
E-TAppTAbs says that when we find a type application is applied to a type abstraction, we carry out the a substitution wherever that type appears in the term  $t_2$ 

(E-TAPP)

out the a substitution wherever that type appears in the term  $t_{i2}$ .

$$\frac{}{(\lambda\;X.t_{12})[T_2]\longrightarrow [X\mapsto T_2]\,t_{12}} \qquad \text{(E-TAPPTABS)}$$
 In this case that will be in the substitutions will be in the type annotations on the lambda

 $\frac{t_1 \longrightarrow t_1{}'}{t_1 \ [T_2] \longrightarrow t_1{}' \ [T_2]}$ 

(E-TAPP)

abstractions.

(E-TAPP) (E-TAPPTABS)

$$\frac{t_1 \longrightarrow t_1{}'}{t_1 \ [\mathcal{T}_2] \longrightarrow t_1{}' \ [\mathcal{T}_2]}$$

$$\frac{(\lambda X.t_{12})[T_2] \longrightarrow [X \mapsto T_2]t_{12}}{(\lambda X.t_{12})[T_2] \longrightarrow [X \mapsto T_2]t_{12}}$$

$$\mapsto T_2]t_1$$



For example:  $(\lambda X . \lambda x : X . x)$  [Int]  $\longrightarrow \lambda x : Int . x$ 

### Typing rules (old)

$$\overline{\Gamma \vdash x:T}$$

 $x:T\in\Gamma$ 

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

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

(T-VAR)

(T-ABS)

$$\Gamma \vdash t_1 \ t_2 : T_2$$

$$\Gamma, x : T_1 \vdash t : T_2$$

Γ,	X	$\vdash$	<i>t</i> <sub>2</sub> :	$T_2$
$\Gamma \vdash (\lambda$	Χ	. t <sub>2</sub>	):∀	Χ

 $\Gamma \vdash t_1[T_2] : [X \mapsto T_2]T_{12}$ T-TAbs gives type abstractions a universal type.

 $\Gamma \vdash t_1 : \forall X . T_{12}$ 

$\Gamma \vdash (\lambda \mid X \mid t_2) : \forall \mid X \mid T_2$
$\Gamma \vdash t_1 : \forall X . T_{12}$
$\Gamma \vdash t_1[T_2] : [X \mapsto T_2]T_{12}$

 $T_{2}T_{12}$  (T-TAPP)

(T-TABS)

The type variable binding in the term becomes the binding in the universal type.

 $\Gamma, X \vdash t_2 : T_2$ 

Γ,	X	$\vdash$	<i>t</i> <sub>2</sub> :	$T_2$
$\Gamma \vdash (\lambda$	Χ	. t <sub>2</sub>	):∀	Χ

 $\begin{array}{ccccc}
 & (\lambda & X \cdot t_2) : \forall & X \cdot T_2 \\
 & \Gamma \vdash t_1 : \forall & X \cdot T_{12}
\end{array}$ 

 $\Gamma \vdash t_1[T_2] : [X \mapsto T_2]T_{12}$ 

(T-TABS)

(T-TAPP)

Other than that, the type doesn't change.

	$\Gamma \vdash (\lambda \mid X \cdot t_2): \forall \mid X \cdot T_2$	(1-1Abs)
	$rac{\Gamma \vdash \ t_1 : orall \ X \ . \ T_{12}}{\Gamma \vdash \ t_1[T_2] : [X \mapsto T_2] T_{12}}$	(Т-ТАРР)
٦	The type variable is mentioned in the context, but we're not doing much w	ith it (for now).

 $\Gamma, X \vdash t_2 : T_2$ 

(T-TABS)

The type variable is mentioned in the context, but we're not doing much with it (for now)

Γ,	Χ	$\vdash$	$t_2$ :	$T_2$
<b>Γ</b> ⊢ (λ	Χ	. t <sub>2</sub>	):∀	X

 $\frac{2}{X \cdot T_2}$ 

(T-TABS)

(T-TAPP)

 $\frac{\Gamma \vdash t_1 : \forall X . T_{12}}{\Gamma \vdash t_1[T_2] : [X \mapsto T_2]T_{12}}$ 

T-TApp consumes the universal types.

Γ,	X	$\vdash$	<i>t</i> <sub>2</sub> :	$T_2$	
<b>Γ</b> ⊢ (λ	X	. t <sub>2</sub>	):∀	<i>X</i> .	7
Γ⊢	$t_1$	: ∀	<i>X</i> .	$T_{12}$	

 $\Gamma \vdash (\lambda \mid X \cdot t_2) : \forall \mid X \cdot T_2$  $\Gamma \vdash t_1 : \forall \mid X \cdot T_{12}$  $\Gamma \vdash t_1[T_2] : [X \mapsto T_2]T_{12}$ (T-TAPP)

(T-TABS)

We start with a term with a universal type.

Γ,	X	$\vdash$	<i>t</i> <sub>2</sub> :	$T_2$
<b>Γ</b> ⊢ (λ	X	. t <sub>2</sub>	):∀	X

 $\Gamma \vdash t_1 : \forall X . T_{12}$ 

(T-TABS) . T<sub>2</sub>

 $\overline{\Gamma \vdash t_1[T_2] : [X \mapsto T_2] T_{12}}$ We apply a type to that term.

(T-TAPP)

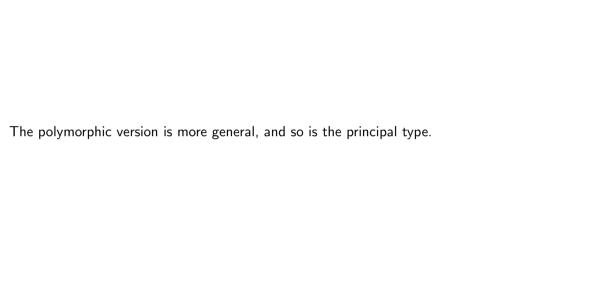
	Γ, <i>X</i>	$\vdash$	$t_2$ :	$T_2$
<u>Γ⊢ (</u> ,	$\lambda X$	. t <sub>2</sub>	):∀	<i>X</i> .
Γ	$\vdash t_1$	: ∀	<i>X</i> .	$T_{12}$

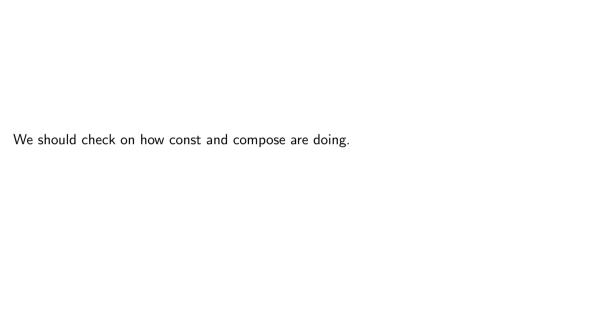
 $\overline{\Gamma \vdash t_1[T_2] : [X \mapsto T_2] T_{12}}$ 

And we get the type substitution that we were after.

We are now in a place where we have principal types rather than unique types.

We can treat the type of the id function in $(\lambda X.\lambda x:X.x)$ true as either $\forall A.A \rightarrow A$ or	
Bool → Bool	





 $\lambda X \cdot \lambda Y$ .

 $\lambda x : X . \lambda y : Y .$ 

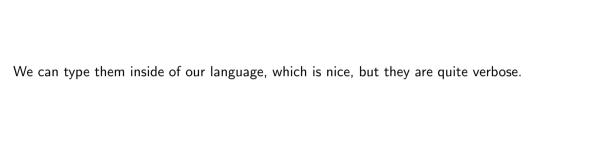
 $: \forall A . \forall B . A \rightarrow B \rightarrow A$ 

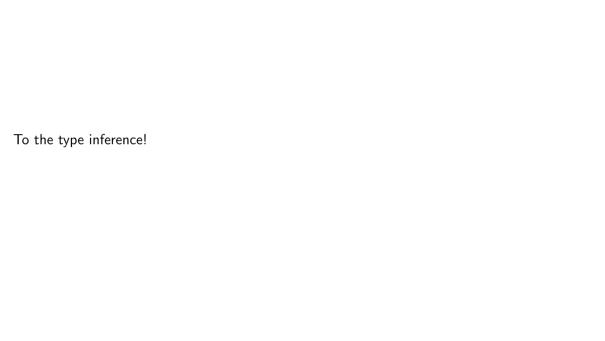
 $\lambda X \cdot \lambda Y \cdot \lambda Z$ .

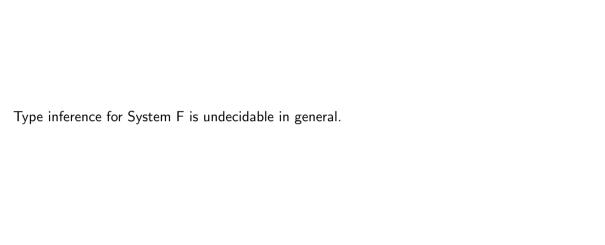
f(g x)

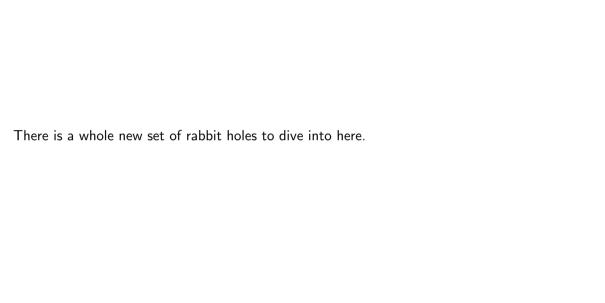
 $: \forall A . \forall B . \forall C . (B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C)$ 

$$\lambda \ f: Y \rightarrow Z \ . \ \lambda \ g: X \rightarrow Y \ . \ \lambda \ x: X \ .$$

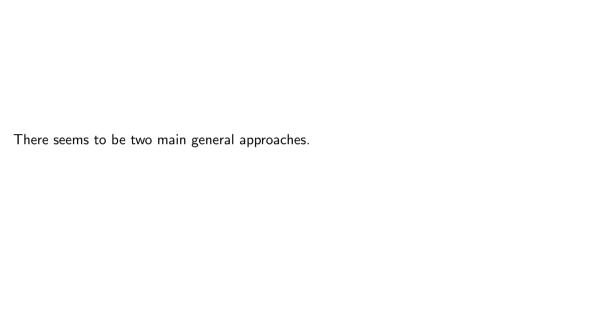








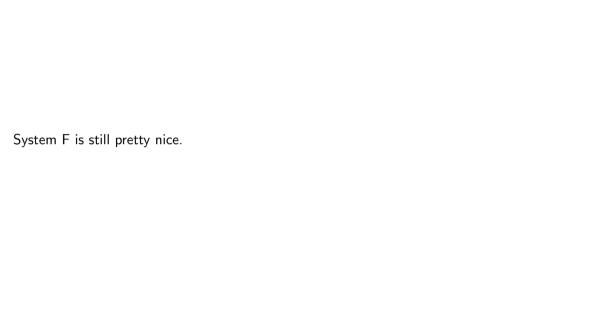
The various options differ in whether they deal with higher rank types and impredicativity, and also in where you need to put annotations and how predictable the need for annotations is.

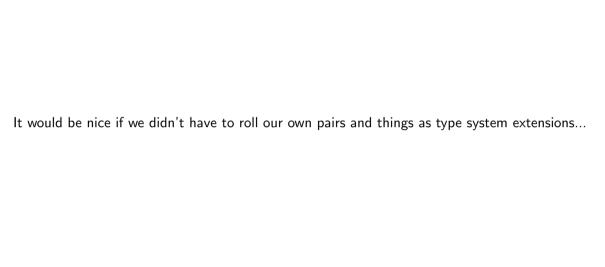


1. Use fancier type schemes - see ML<sup>F</sup>, HMF, HML.

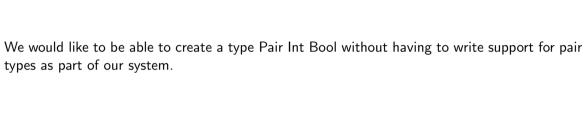
2. Use bidirectional type checking / local type inference - make checking and inference mutually recursive so that you can propagate information from annotations to places where it

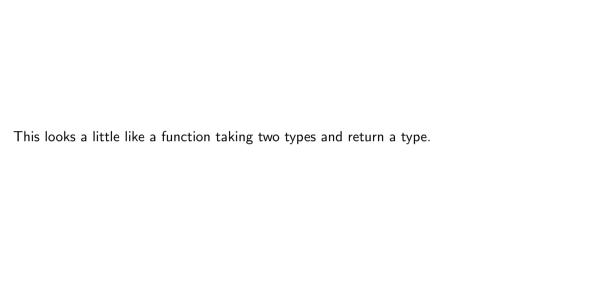
might be needed.

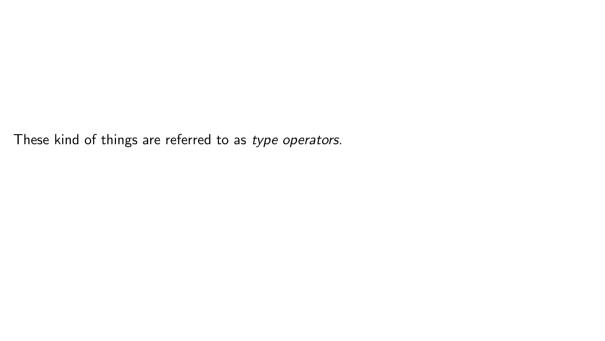




# Universal types and type operators: System $F_{\omega}$







# $T \rightarrow T$ $\forall X :: K . T$ $\lambda X::K.T$ K kind of proper types $K \Rightarrow K$ kind of type operators

Types and kinds

type variable function arrow univeral type operator abstraction operator application

We add operator abstraction to our set of types.

#### type variable $T \rightarrow T$ function arrow ∀ X ::K .T univeral type $\lambda X::K.T$ operator abstraction operator application K kind of proper types $K \Rightarrow K$ kind of type operators

Types and kinds

That presents a need for operator application in order to use it.

#### type variable $T \rightarrow T$ function arrow $\forall X :: K . T$ univeral type $\lambda X::K.T$ operator abstraction operator application K kind of proper types $K \Rightarrow K$ kind of type operators

Types and kinds

We add in a kind system in order to check that our types make sense.

#### Types and kinds type variable $T \rightarrow T$ function arrow $\forall X :: K . T$ univeral type $\lambda X::K.T$ operator abstraction operator application K kind of proper types $K \Rightarrow K$ kind of type operators

Pair Int Bool makes sense.

### Types and kinds

T :=	• • •	
	X	type variable
	au o au	function arrow
	$\forall X :: K . T$	univeral type
	$\lambda X :: K.T$	operator abstraction
	TT	operator application
<i>K</i> :=		
	*	kind of proper types
	$K \Rightarrow K$	kind of type operators

It has kind  $* \Rightarrow * \Rightarrow *$ 

### Types and kinds

<i>I</i> :=	• • •	
	X	type variable
	${\mathcal T}  o {\mathcal T}$	function arrow
	$\forall X :: K . T$	univeral type
	$\lambda X :: K.T$	operator abstraction
	TT	operator application
K :=		
	*	kind of proper types
	$K \Rightarrow K$	kind of type operators

Int Bool makes no sense at all.

### Types and kinds

$$X$$
 type variable  $T o T$  function arrow  $\forall X :: K . T$  univeral type  $\lambda X :: K . T$  operator abstraction  $T . T$  operator application  $K :=$  kind of proper types

kind of type operators

 $K \Rightarrow K$ 

It should be ill-kinded and hence ruled out.

#### Terms and values variable X $\lambda \times T.t$ abstraction t t function application $\lambda X :: K .t$ type abstraction t [T] type application $\lambda \times T.t$ abstraction $\lambda X :: K .t$ type abstraction

The terms and values have not changed except for the extra kind annotation.

#### Terms and values variable X $\lambda \times T.t$ abstraction function application t t $\lambda X :: K .t$ type abstraction t [T] type application $\lambda \times T.t$ abstraction $\lambda X :: K .t$ type abstraction

The same is true of the evaluation rules.

### Typing rules (changes from STLC)

$$\frac{\Gamma \vdash T_1 ::* \qquad \Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (\lambda \; x : T_1.t) : T_1 \to T_2} \tag{T-Abs}$$
 We need to make sure that our type annotations in lambda are types, rather than type operators.

 $\frac{x:T\in\Gamma}{\Gamma\vdash x:T}$ 

 $\Gamma \vdash t_1: T_1 \rightarrow T_2 \qquad \Gamma \vdash t_2: T1$ 

 $\Gamma \vdash t_1 \ t_2 : T_2$ 

(T-VAR)

(T-App)

### Typing rules (changes from System F)

$$\frac{\Gamma, X :: \mathcal{K}_1 \vdash t_2 : \mathcal{T}_2}{\Gamma \vdash (\lambda X :: \mathcal{K}_1 \mid t_2) : \forall X :: \mathcal{K}_1 \mid \mathcal{T}_2}$$

$$\Gamma \vdash t_1 : \forall X :: \mathcal{K}_{11} \mid \mathcal{T}_{12} \qquad \Gamma \vdash \mathcal{T}_2 :: \mathcal{K}_{11}$$

We need to make sure that the kinds all match up properly with our universal types.

 $\Gamma \vdash t_1[T_2] : [X \mapsto T_2]T_{12}$ 

(T-TABS)

(T-TAPP)

$$\frac{\Gamma \vdash t:S \qquad S \equiv T \qquad \Gamma \vdash T::*}{\Gamma \vdash t:T}$$

(T-EQ)

If we allow type operators, we can write a type level id

$$\mathsf{Id} = \lambda \; X :: \; * \; . \; X$$

$$\frac{\Gamma \vdash t:S \qquad S \equiv T \qquad \Gamma \vdash T::*}{\Gamma \vdash t:T}$$

 $\mathsf{Nat} \to \mathsf{Bool}$ 

(T-EQ)

Given that, these all mean the same thing:

$$\begin{array}{c} \mathsf{Id} \; \mathsf{Nat} \to \mathsf{Bool} \\ \\ \mathsf{Id} \; \mathsf{Nat} \to \mathsf{Id} \; \mathsf{Bool} \\ \\ \mathsf{Id} \; (\mathsf{Nat} \to \mathsf{Bool}) \end{array}$$

$$\frac{\Gamma \vdash t:S \qquad S \equiv T \qquad \Gamma \vdash T::*}{\Gamma \vdash t:T}$$

(T-EQ)

We need a notion of type equivalence to deal with this.

$$\frac{\Gamma \vdash t:S \qquad S \equiv T \qquad \Gamma \vdash T::*}{\Gamma \vdash t:T}$$

(T-EQ)

 $S \equiv T$  is our type equivalence relationship.

$\overline{T} \equiv \overline{T}$	(Q-REFL)	
$\frac{T \equiv S}{S \equiv T}$	(Q-SYM)	
$\frac{S \equiv T \qquad T \equiv U}{S \equiv U}$	(Q-TRANS)	
$S_1 \equiv \mathcal{T}_1 \qquad S_2 \equiv \mathcal{T}_2$	(O-ARROW)	

 $\overline{S_1 \rightarrow S_2 \equiv T_1 \rightarrow T_2}$ 

Typing equivalence rules (partial)

Grah! It's a non-deterministic set of rules!

$S \equiv T$
$\frac{S \equiv T \qquad T \equiv U}{S \equiv U}$
$rac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{S_1  ightarrow S_2 \equiv T_1  ightarrow T_2}$

 $T \equiv T$ 

 $T \equiv S$ 

Typing equivalence rules (partial)

(Q-TRANS)

(Q-REFL)

(Q-SYM)

(Q-ARROW)

...

We should look at the kind system and see if it helps us out.

Kinding rules	
$T := \dots X$	$\frac{X::K\in\Gamma}{\Gamma\vdash X::K} \qquad \text{(K-TVAR)}$
λ X::K.T Τ Τ	$\frac{\Gamma \vdash \mathcal{T}_1 :: \mathcal{K}_1 \to \mathcal{K}_2 \qquad \Gamma \vdash \mathcal{T}_2 :: \mathcal{K}_1}{\Gamma \vdash \mathcal{T}_1 \ \mathcal{T}_2 :: \mathcal{K}_2} \left( \text{K-App} \right)$
$K := K \Rightarrow K$	$\frac{\Gamma, X :: \mathcal{K}_1 \vdash T_2 : \mathcal{K}_2}{\Gamma \vdash (\lambda \ X :: \mathcal{K}_1 . T_2) :: \mathcal{K}_1 \rightarrow \mathcal{K}_2} \text{ (K-Abs)}$

This looks familiar...

### Kinding rules *X*::*K* ∈ Γ (K-TVAR) $\frac{\Gamma \vdash T_1 :: K_1 \to K_2 \qquad \Gamma \vdash T_2 :: K_1}{\Gamma \vdash T_1 \quad T_2 :: K_2} \text{ (K-APP)}$ $\lambda X::K.T$ T T $\frac{\Gamma, X :: K_1 \vdash T_2 : K_2}{\Gamma \vdash (\lambda X :: K_1, T_2) :: K_1 \rightarrow K_2}$ (K-Abs) $K \Rightarrow K$

It's simply typed lambda calculus, raised up one level!

### Kinding rules *X*::*K* ∈ Γ (K-TVAR) $\frac{\Gamma \vdash T_1 :: K_1 \to K_2 \qquad \Gamma \vdash T_2 :: K_1}{\Gamma \vdash T_1 \quad T_2 :: K_2} \text{ (K-APP)}$ $\lambda X::K.T$ T $\frac{\Gamma, X :: K_1 \vdash T_2 : K_2}{\Gamma \vdash (\lambda X :: K_1, T_2) :: K_1 \rightarrow K_2}$ (K-Abs) $K \Rightarrow K$

That gives us high confidence in using our kind system to keep the types in check.

# 

*X*::*K* ∈ Γ

(K-TVAR)

Kinding rules

Kinding rules	
$T := \dots X$	$\frac{X::K\in\Gamma}{\Gamma\vdash X::K} \qquad \text{(K-TVAR)}$
λ X::K.T Τ Τ	$\frac{\Gamma \vdash T_1 :: \mathcal{K}_1 \rightarrow \mathcal{K}_2 \qquad \Gamma \vdash T_2 :: \mathcal{K}_1}{\Gamma \vdash T_1 \ T_2 :: \mathcal{K}_2} \left( \text{K-App} \right)$
$\mathcal{K}$ := $\mathcal{K} \Rightarrow \mathcal{K}$	$\frac{\Gamma, X :: \mathcal{K}_1 \vdash T_2 : \mathcal{K}_2}{\Gamma \vdash (\lambda \ X :: \mathcal{K}_1 . T_2) :: \mathcal{K}_1 \rightarrow \mathcal{K}_2} \text{ (K-Abs)}$

On top of all that, it provides a hint about how to deal with the type equivalence problem.

### Kinding rules *X*::*K* ∈ Γ (K-TVAR) $\frac{\Gamma \vdash T_1 :: K_1 \to K_2 \qquad \Gamma \vdash T_2 :: K_1}{\Gamma \vdash T_1 \quad T_2 :: K_2} \text{ (K-APP)}$ $\lambda X::K.T$ T $\frac{\Gamma, X :: K_1 \vdash T_2 : K_2}{\Gamma \vdash (\lambda X :: K_1 . T_2) :: K_1 \rightarrow K_2}$ (K-Abs) $K \Rightarrow K$

STLC is strongly normalizing...

# $\frac{\Gamma \vdash T_1 :: K_1 \to K_2 \qquad \Gamma \vdash T_2 :: K_1}{\Gamma \vdash T_1 \quad T_2 :: K_2} \text{ (K-APP)}$ $\lambda X::K.T$ $\frac{\Gamma, X :: K_1 \vdash T_2 : K_2}{\Gamma \vdash (\lambda X :: K_1, T_2) :: K_1 \rightarrow K_2}$ (K-Abs) $K \Rightarrow K$ so we can use STLC evaluation rules to normalize our types before we compare them, giving

*X*::*K* ∈ Γ

(K-TVAR)

Kinding rules

us algorithmic type equivalence.

#### Kinding rules

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \to T_2 :: *}$$

 $\frac{\Gamma, X :: K_1 \vdash T_2 :: *}{\Gamma \vdash \forall X :: K_1 . T_2 :: *}$ 

(K-Arrow)

(K-ALL)

We need a few more kinding rules to sanity check the other features in our type system.

### Kinding rules

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \rightarrow T_2 :: *}$$

 $\frac{\Gamma, X :: K_1 \vdash T_2 :: *}{\Gamma \vdash \forall X :: K_1 . T_2 :: *}$ 

(K-Arrow)

(K-ALL)

Neither of these mess with the strong normalization property

# Kinding rules

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \rightarrow T_2 :: *}$$

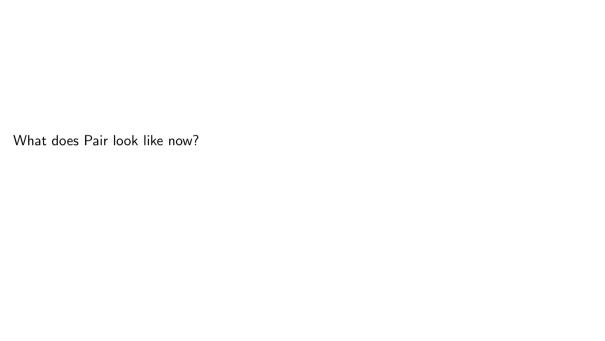
 $\frac{\Gamma, X :: K_1 \vdash T_2 ::*}{\Gamma \vdash \forall X :: K_1 ... T_2 ::*}$ 

(K-Arrow)

(K-All)

Hurrah!

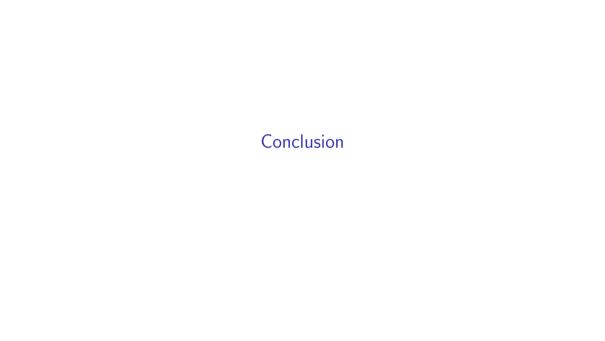
Hurra

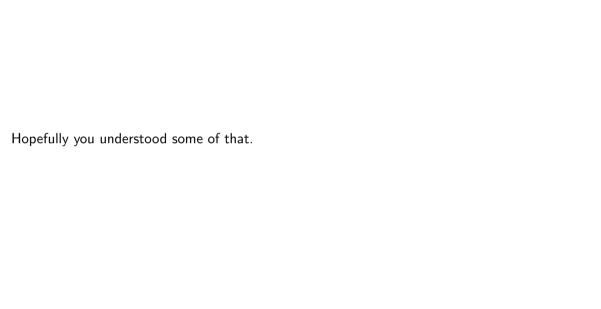


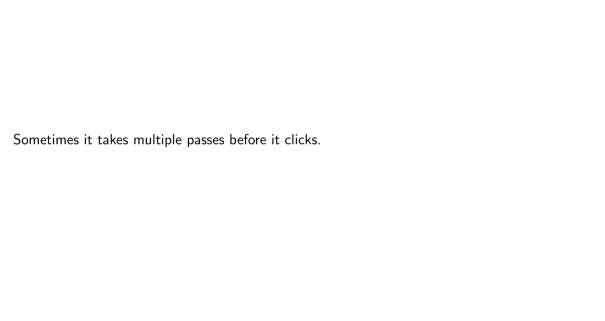


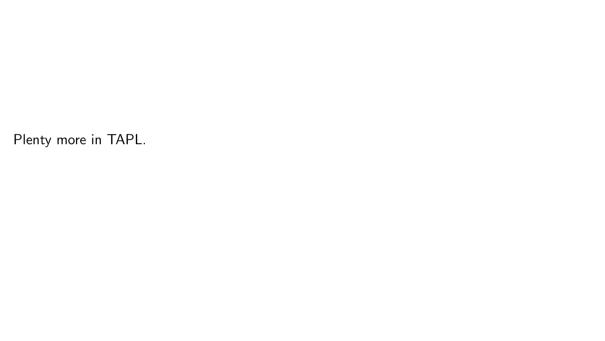
(Let us assume that we have set up a basic record system)

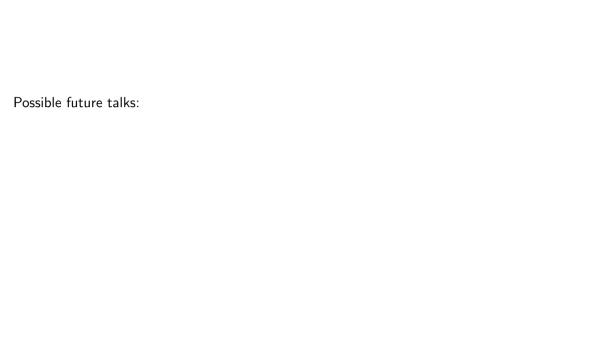
 $\lambda \ A:: *.\lambda \ B:: *.\{ \mathsf{fst} : A, \mathsf{snd} : B \}$ 











▶ Implementing interpreters and type systems for these (and other) languages.

- ▶ Implementing interpreters and type systems for these (and other) languages.
- ► Funkier features: type classes, row polymorphism, linear types, and other fun.

- ▶ Implementing interpreters and type systems for these (and other) languages.
- ► Funkier features: type classes, row polymorphism, linear types, and other fun.
- Funkier reacures: type classes, row polymorphism, linear types, and other run.

▶ How to write compilers for these languages with LLVM.

- ▶ Implementing interpreters and type systems for these (and other) languages.
- ► Funkier features: type classes, row polymorphism, linear types, and other fun.
- ▶ How to write compilers for these languages with LLVM.

More about what you can do with unification.

From to write compliers for these languages with ELVIV