Programming Language Theory

Dave Laing

BFPG February 2019

Introduction

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

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

Very useful to know if you want to read certain types of blog posts and papers.

You only need to know about a few things before you can read *many* more papers.

PLT is not as scary as it seems...

… but it can seem quite scary at first.

Terms, values and types

t	$:=$		
		x	*variable*
		$\lambda\ x{:}T.t$	*abstraction*
		$t\ t$	*function application*
v	$:=$		
		$\lambda\ x{:}T.t$	*abstraction*
T	$:=$		
		$T \to T$	*function arrow*

Small-step semantics

$$\frac{t_1 \longrightarrow {t_1}'}{t_1\ t_2 \longrightarrow {t_1}'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow {t_2}'}{v_1\ t_2 \longrightarrow v_1\ {t_2}'} \qquad \text{(E-App2)}$$

$$\frac{}{(\lambda\ x{:}T.t_1)t_2 \longrightarrow [x \mapsto t_2]\, t_1} \qquad \text{(E-AppAbs)}$$

Typing rules

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x{:}T} \qquad \text{(T-Var)}$$

$$\frac{\Gamma \vdash t_1{:}T_1 \to T_2 \qquad \Gamma \vdash t_2{:}T1}{\Gamma \vdash t_1\ t_2{:}T_2} \qquad \text{(T-App)}$$

$$\frac{\Gamma, x{:}T_1 \vdash T_2}{\Gamma \vdash (\lambda\ x{:}T_1.t){:}T_1 \to T_2} \qquad \text{(T-Abs)}$$

These talks are partly about getting you started with the notation, concepts and terminology.

They will also cover how a lot of common language features are defined.

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

Natural deduction rules

We use natural deduction style rules for most of this.

$$\frac{\mathit{Assumption}_1 \qquad \mathit{Assumption}_2 \qquad \ldots}{\mathit{Conclusion}} \qquad (\textsc{Rule-Name})$$

Rules for evenness

$$\frac{}{\textsf{even}\ 0} \quad (\textsc{Even-Zero})$$

$$\frac{\textsf{even}\ x}{\textsf{even}\ \ (x+2)} \quad (\textsc{Even-Add})$$

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

Rules for evenness

$$\frac{}{\textsf{even } 0} \qquad (\textsc{Even-Zero})$$

$$\frac{\textsf{even } x}{\textsf{even } (x+2)} \qquad (\textsc{Even-Add})$$

Some rules - like Even-Add - are recursive.

Rules for evenness

$$\frac{}{\textsf{even } 0} \qquad (\textsc{Even-Zero})$$

$$\frac{\textsf{even } x}{\textsf{even } (x+2)} \qquad (\textsc{Even-Add})$$

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

Rules for evenness

$$\frac{}{\textsf{even}\ 0} \qquad (\textsc{Even-Zero})$$

$$\frac{\textsf{even}\ x}{\textsf{even}\ (x+2)} \qquad (\textsc{Even-Add})$$

Usually only one rule will apply at one time, and so the order we organize these rules in doesn't matter.

Rules for evenness

$$\frac{}{\textsf{even } 0} \quad (\textsc{Even-Zero})$$

$$\frac{\textsf{even } x}{\textsf{even } (x+2)} \quad (\textsc{Even-Add})$$

We are also dealing with an "open world".

Rules for evenness

$$\frac{}{\textsf{even } 0} \quad (\textsc{Even-Zero})$$

$$\frac{\textsf{even } x}{\textsf{even } (x+2)} \quad (\textsc{Even-Add})$$

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

Rules for evenness, now with bonus rules

$$\frac{}{\textsf{even } 0} \quad (\textsc{Even-Zero})$$

$$\frac{\textsf{even } x}{\textsf{even } (x+2)} \quad (\textsc{Even-Add})$$

$$\frac{\textsf{odd } x}{\textsf{even } (x+1)} \quad (\textsc{Even-Odd})$$

$$\frac{\textsf{even } x}{\textsf{odd } (x+1)} \quad (\textsc{Odd-Even})$$

Let us add some more rules.

Rules for evenness, now with bonus rules

$$\frac{}{\textsf{even } 0} \qquad (\textsc{Even-Zero})$$

$$\frac{\textsf{even } x}{\textsf{even } (x+2)} \qquad (\textsc{Even-Add})$$

$$\frac{\textsf{odd } x}{\textsf{even } (x+1)} \qquad (\textsc{Even-Odd})$$

$$\frac{\textsf{even } x}{\textsf{odd } (x+1)} \qquad (\textsc{Odd-Even})$$

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

Rules for evenness, now with bonus rules

$$\frac{}{\textsf{even } 0} \quad (\textsc{Even-Zero})$$

$$\frac{\textsf{even } x}{\textsf{even } (x+2)} \quad (\textsc{Even-Add})$$

$$\frac{\textsf{odd } x}{\textsf{even } (x+1)} \quad (\textsc{Even-Odd})$$

$$\frac{\textsf{even } x}{\textsf{odd } (x+1)} \quad (\textsc{Odd-Even})$$

We want the rules to be *deterministic* - the answers through all of the paths agree, and the paths are finite.

Equivalence relations

$$\frac{}{P = P} \quad \text{(Reflexivity)}$$

$$\frac{Q = P}{P = Q} \quad \text{(Symmetry)}$$

$$\frac{P = Q \qquad Q = R}{P = R} \quad \text{(Transitivity)}$$

Sometimes we have rules like Symmetry, which could be applied over and over and spin forever.

Equivalence relations

$$\frac{}{P = P} \quad \text{(Reflexivity)}$$

$$\frac{Q = P}{P = Q} \quad \text{(Symmetry)}$$

$$\frac{P = Q \qquad Q = R}{P = R} \quad \text{(Transitivity)}$$

Those rules usually exist in that form to explain why a system has some desired properties, or to assist with proofs.

Equivalence relations

$$\frac{}{P = P} \quad \text{(Reflexivity)}$$

$$\frac{Q = P}{P = Q} \quad \text{(Symmetry)}$$

$$\frac{P = Q \qquad Q = R}{P = R} \quad \text{(Transitivity)}$$

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

Equivalence relations

$$\frac{}{P = P} \quad (\textsc{Reflexivity})$$

$$\frac{Q = P}{P = Q} \quad (\textsc{Symmetry})$$

$$\frac{P = Q \qquad Q = R}{P = R} \quad (\textsc{Transitivity})$$

This is usually proved by a proof of logical equivalence of the non-deterministic and algorithmic rules sets.

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

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

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

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

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

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

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

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

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

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

```haskell
type RuleSet a b = a -> Maybe b
type RuleSet a b = a -> Maybe b

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

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

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

```haskell
type RuleSet a b = a -> Maybe b

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

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

mkRuleSet rules =
  let
    ruleSet a =
      asum .
      fmap (\r -> r ruleSet a) $
      rules
  in
    ruleSet
```

```haskell
evenZero :: Rule Int ()
evenZero _ x
  | x == 0 = Just ()
  | otherwise = Nothing
```

$$\frac{}{\textsf{even } 0} \quad (\textsc{Even-Zero})$$

```haskell
evenAddTwo :: Rule Int ()
evenAddTwo e n
  | n >= 2 = e (n - 2)
  | otherwise = Nothing
```

$$\frac{\mathsf{even}\ x}{\mathsf{even}\ \ (x+2)}\quad(\textsc{Even-Add})$$

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

```haskell
evenOdd :: RuleSet Int () -> Rule Int ()
evenOdd o _ n
  | n >= 1 = o (n - 1)
  | otherwise = Nothing
```

$$\frac{\mathsf{odd}\ x}{\mathsf{even}\ \ (x+1)}\quad(\textsc{Even-Odd})$$

```haskell
oddEven :: RuleSet Int () -> Rule Int ()
oddEven e _ n
  | n >= 1 = e (n - 1)
  | otherwise = Nothing
```

$$\frac{\textsf{even}\ x}{\textsf{odd}\ \ (x+1)} \quad (\textsc{Odd-Even})$$

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

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

```
> evenR 3
> evenR 3
Nothing
```

```
> evenR 3
Nothing
> evenR 4
> evenR 3
Nothing
> evenR 4
Just ()
```

```
> evenR 3
Nothing
> evenR 4
Just ()
> oddR 3
> evenR 3
Nothing
> evenR 4
Just ()
> oddR 3
Just ()
```

```
> evenR 3
Nothing
> evenR 4
Just ()
> oddR 3
Just ()
> oddR 4
> evenR 3
Nothing
> evenR 4
Just ()
> oddR 3
Just ()
> oddR 4
Nothing
```

Integers

Let us look at a simple language, starting with the terms.

Terms

t	$:=$	
	$\langle\text{int}\rangle$	*constant integer*
	$t + t$	*addition*

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

Terms

t	$:=$		
		$\langle \text{int} \rangle$	*constant integer*
		$t + t$	*addition*

This is the abstract syntax of our language.

3

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

```haskell
data Term =
    TmInt Int
  | TmAdd Term Term
  deriving (Eq, Ord, Show)
```

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

```
TmInt 3
```

3

```
TmAdd (TmAdd (TmInt 1) (TmInt 2)) (TmInt 3)
```

$(1 + 2) + 3$

What are the values for the Integer language?

By definitions:

$$
\begin{array}{lll}
v & := & \\
 & \langle \mathrm{int} \rangle & \textit{constant integer}
\end{array}
$$

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

By rules:

$$\frac{}{\textsf{value}\ \langle\textsf{int}\rangle}\quad(\textsc{V-Int})$$

Having the rules can help with the implementation.

When we e*valu*ate a term, we are turning it into a *value*.

Values are specified as part of the *syntax* of a languge.

Evaluation proceeds in *steps*.

The set of steps gives us the *small-step semantics* for the language.

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

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

Small-step semantics (partial)

$$\frac{}{\langle \mathsf{int}_1 \rangle + \langle \mathsf{int}_2 \rangle \longrightarrow \langle \mathsf{int}_1 + \mathsf{int}_2 \rangle} \quad (\textsc{E-AddInt})$$

E-AddInt does the actual addition.

Small-step semantics (partial)

$$\frac{t_1 \longrightarrow {t_1}'}{t_1 + t_2 \longrightarrow {t_1}' + t_2} \qquad \text{(E-ADD1)}$$

$$\frac{\text{value } t_1 \qquad t_2 \longrightarrow {t_2}'}{t_1 + t_2 \longrightarrow t_1 + {t_2}'} \qquad \text{(E-ADD2)}$$

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

Small-step semantics (partial)

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

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

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

Small-step semantics

$$\frac{t_1 \longrightarrow {t_1}'}{t_1 + t_2 \longrightarrow {t_1}' + t_2} \qquad (\textsc{E-Add1})$$

$$\frac{t_2 \longrightarrow {t_2}'}{v_1 + t_2 \longrightarrow v_1 + {t_2}'} \qquad (\textsc{E-Add2})$$

$$\frac{}{\langle \mathsf{int}_1 \rangle + \langle \mathsf{int}_2 \rangle \longrightarrow \langle \mathsf{int}_1 + \mathsf{int}_2 \rangle} \qquad (\textsc{E-AddInt})$$

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

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

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

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

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

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

A language can be *normalizing*: there is an evaluation order that means that finite-sized terms will always evaluate in finite time.

A language can be *strongly normalizing*: for any evaluation order, finite-sized terms will always evaluate in finite time.

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

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

$$\frac{}{v \Rightarrow v} \qquad \text{(Big-Value)}$$

$$\frac{t \longrightarrow t' \qquad t' \Rightarrow v}{t \Rightarrow v} \qquad \text{(Big-Step)}$$

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

The complete evaluation takes three steps.

First:

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

Then:

$$\dfrac{\dfrac{}{3+4 \longrightarrow 7}\ \text{E-AddInt}}{3 + \left(3+4\right) \longrightarrow 3 + 7}\ \text{E-Add2}$$

Finally:

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

The complete evaluation takes three steps.

First:

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

Then:

$$\dfrac{\dfrac{}{3+4 \longrightarrow 7}\ \text{E-AddInt}}{3 + \left(3+4\right) \longrightarrow 3 + 7}\ \text{E-Add2}$$

Finally:

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

The complete evaluation takes three steps.

First:

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

Then:

$$\dfrac{\dfrac{}{3+4 \longrightarrow 7}\ \text{E-AddInt}}{3 + \left(3+4\right) \longrightarrow 3 + 7}\ \text{E-Add2}$$

Finally:

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

The complete evaluation takes three steps.

First:

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

Then:

$$\dfrac{\dfrac{}{3+4 \longrightarrow 7}\ \text{E-AddInt}}{3+\left(3+4\right) \longrightarrow 3+7}\ \text{E-Add2}$$

Finally:

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

The complete evaluation takes three steps.

First:

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

Then:

$$\dfrac{\dfrac{}{3+4 \longrightarrow 7}\ \textsf{E-AddInt}}{3+\left(3+4\right) \longrightarrow 3+7}\ \textsf{E-Add2}$$

Finally:

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

The complete evaluation takes three steps.

First:

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

Then:

$$\dfrac{\dfrac{}{3+4 \longrightarrow 7}\ \text{E-AddInt}}{3 + (3+4) \longrightarrow 3 + 7}\ \text{E-Add2}$$

Finally:

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

The complete evaluation takes three steps.

First:

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

Then:

$$\dfrac{\dfrac{}{3+4 \longrightarrow 7}\ \text{E-AddInt}}{3+\left(3+4\right) \longrightarrow 3+7}\ \text{E-Add2}$$

Finally:

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

The complete evaluation takes three steps.

First:

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

Then:

$$\dfrac{\dfrac{}{3+4 \longrightarrow 7}\ \text{E-AddInt}}{3+\left(3+4\right) \longrightarrow 3+7}\ \text{E-Add2}$$

Finally:

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

The complete evaluation takes three steps.

First:

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

Then:

$$\dfrac{\dfrac{}{3+4 \longrightarrow 7}\;\text{E-AddInt}}{3 + \left(3+4\right) \longrightarrow 3 + 7}\;\text{E-Add2}$$

Finally:

$$\dfrac{}{3+7 \longrightarrow 10}\;\text{E-AddInt}$$

The complete evaluation takes three steps.

First:

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

Then:

$$\dfrac{\dfrac{}{3+4 \longrightarrow 7}\;\text{E-AddInt}}{3+\left(3+4\right) \longrightarrow 3+7}\;\text{E-Add2}$$

Finally:

$$\dfrac{}{3+7 \longrightarrow 10}\;\text{E-AddInt}$$

The complete evaluation takes three steps.

First:

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

Then:

$$\dfrac{\dfrac{}{3+4 \longrightarrow 7}\;\text{E-AddInt}}{3+\left(3+4\right) \longrightarrow 3+7}\;\text{E-Add2}$$

Finally:

$$\dfrac{}{3+7 \longrightarrow 10}\;\text{E-AddInt}$$

The complete evaluation takes three steps.

First:

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

Then:

$$\dfrac{\dfrac{}{3+4 \longrightarrow 7}\ \text{E-AddInt}}{3+\left(3+4\right) \longrightarrow 3+7}\ \text{E-Add2}$$

Finally:

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

The complete evaluation takes three steps.

First:

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

Then:

$$\dfrac{\dfrac{}{3+4 \longrightarrow 7}\ \text{E-AddInt}}{3+\left(3+4\right) \longrightarrow 3+7}\ \text{E-Add2}$$

Finally:

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

The complete evaluation takes three steps.

First:

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

Then:

$$\dfrac{\dfrac{}{3+4 \longrightarrow 7}\ \text{E-AddInt}}{3+\left(3+4\right) \longrightarrow 3+7}\ \text{E-Add2}$$

Finally:

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

The complete evaluation takes three steps.

First:

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

Then:

$$\frac{\dfrac{}{3+4 \longrightarrow 7}\;\text{E-AddInt}}{3+\left(3+4\right) \longrightarrow 3+7}\;\text{E-Add2}$$

Finally:

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

The complete evaluation takes three steps.

First:

$$\dfrac{\dfrac{}{1 + 2 \longrightarrow 3}\,\textsf{E-AddInt}}{\left(1 + 2\right) + \left(3 + 4\right) \longrightarrow 3 + \left(3 + 4\right)}\,\textsf{E-Add1}$$

Then:

$$\dfrac{\dfrac{}{3 + 4 \longrightarrow 7}\,\textsf{E-AddInt}}{3 + \left(3 + 4\right) \longrightarrow 3 + 7}\,\textsf{E-Add2}$$

Finally:

$$\dfrac{}{3 + 7 \longrightarrow 10}\,\textsf{E-AddInt}$$

The complete evaluation takes three steps.

First:

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

Then:

$$\dfrac{\dfrac{}{3 + 4 \longrightarrow 7}\ \text{E-AddInt}}{3 + \left(3 + 4\right) \longrightarrow 3 + 7}\ \text{E-Add2}$$

Finally:

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

The complete evaluation takes three steps.

First:

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

Then:

$$\dfrac{\dfrac{}{3+4 \longrightarrow 7}\;\text{E-AddInt}}{3 + \left(3+4\right) \longrightarrow 3 + 7}\;\text{E-Add2}$$

Finally:

$$\dfrac{}{3+7 \longrightarrow 10}\;\text{E-AddInt}$$

```haskell
vInt :: Rule Term ()
vInt _ (TmInt _) =
  Just ()
intValue _ _ =
  Nothing
```

$$\frac{}{\textsf{value}\ \ \langle\textsf{int}\rangle}\qquad(\textsc{V-Int})$$

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

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

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

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

```haskell
eAdd1 :: Rule Term Term
eAdd1 step (TmAdd tm1 tm2) = do
  tm1' <- step tm1
  pure $ TmAdd tm1' tm2
eAdd1 _ _ =
  Nothing
```

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

```haskell
eAdd2 :: RuleSet Term () -> Rule Term Term
eAdd2 value step (TmAdd tm1 tm2) = do
  _ <- value tm1
  tm2' <- step tm2
  pure $ TmAdd tm1 tm2'
eAdd2 _ _ _ =
  Nothing
```

$$\frac{t_2 \longrightarrow {t_2}'}{v_1 + t_2 \longrightarrow v_1 + {t_2}'} \quad (\text{E-ADD2})$$

```haskell
eAdd2 :: Rule Term Term
eAdd2 step (TmAdd tm1@(TmInt _) tm2) = do
  tm2' <- step tm2
  pure $ TmAdd tm1 tm2'
eAdd2 _ _ =
  Nothing
```

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

```
eAddInt :: Rule Term Term
eAddInt _ (TmAdd (TmInt i1) (TmInt i2)) =
  Just $ TmInt (i1 + i2)
eAddInt _ _ =
  Nothing
```

$$\frac{}{\langle \mathsf{int}_1 \rangle + \langle \mathsf{int}_2 \rangle \longrightarrow \langle \mathsf{int}_1 + \mathsf{int}_2 \rangle}\ (\textsc{E-AddInt})$$

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

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

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

> stepR tm

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

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

> stepR tm

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

> stepR >=> stepR $ tm

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

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

> stepR tm

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

> stepR >=> stepR $ tm

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

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

> stepR tm

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

> stepR >=> stepR $ tm

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

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

Just (TmInt 10)
```

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

> stepR tm

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

> stepR >=> stepR $ tm

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

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

Just (TmInt 10)

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

> stepR tm

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

> stepR >=> stepR $ tm

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

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

Just (TmInt 10)

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

Nothing
```

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

eval :: Term -> Term
eval =
  iterR stepR
```

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

Terms are not stuck

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

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

Terms are not stuck

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

For all terms,

the term is either a value or

is able to step .

Terms are not stuck

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

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

Terms are not stuck

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

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

Terms are not stuck

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

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

Terms are not stuck

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

For all terms,

the term is either a value or

is able to step.

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

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

```haskell
exactlyOne :: (Show a)
           => Gen a
           -> RuleSet a b
           -> RuleSet a c
           -> Property
exactlyOne ga r1 r2 = property $ do
  a <- forAll ga
  isJust (r1 a) /== isJust (r2 a)
```

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

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

Booleans

Terms and values

t	$:=$		
		false	*constant false*
		true	*constant true*
		t or t	*disjunction*
v	$:=$		
		false	*false value*
		true	*true value*

Small-step semantics (eager)

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

$$\frac{t_1 \longrightarrow t_1{}'}{t_1 \ \textsf{or}\ t_2 \longrightarrow t_1{}' \ \textsf{or}\ t_2}\ (\textsc{E-Or1}) \qquad \frac{}{\textsf{false or true} \longrightarrow \textsf{true}}\ (\textsc{E-OrFalseTrue})$$

$$\frac{t_2 \longrightarrow t_2{}'}{v_1 \ \textsf{or}\ t_2 \longrightarrow v_1 \ \textsf{or}\ t_2{}'}\ (\textsc{E-Or2}) \qquad \frac{}{\textsf{true or false} \longrightarrow \textsf{true}}\ (\textsc{E-OrTrueFalse})$$

$$\frac{}{\textsf{true or true} \longrightarrow \textsf{true}}\ (\textsc{E-OrTrueTrue})$$

Small-step semantics (lazy)

$$\frac{t_1 \longrightarrow t_1{}'}{t_1 \text{ or } t_2 \longrightarrow t_1{}' \text{ or } t_2} \quad \text{(E-Or1)}$$

$$\frac{}{\text{false or } t_2 \longrightarrow t_2} \quad \text{(E-OrFalse)}$$

$$\frac{}{\text{true or } t_2 \longrightarrow \text{true}} \quad \text{(E-OrTrue)}$$

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

```haskell
vFalse :: Rule Term ()
vFalse _ TmFalse =
  Just ()
vFalse _ _ =
  Nothing
```

$$\frac{}{\textsf{value false}} \quad (\text{V-}\textsc{False})$$

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

$$\frac{}{\textsf{value true}} \quad (\textsc{V-True})$$

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

```haskell
eOr1 :: Rule Term Term
eOr1 step (TmOr tm1 tm2) = do
  tm1' <- step tm1
  pure $ TmOr tm1' tm2
eOr1 _ _ =
  Nothing
```

$$\frac{t_1 \longrightarrow {t_1}'}{t_1 \text{ or } t_2 \longrightarrow {t_1}' \text{ or } t_2} \quad (\textsc{E-Or1})$$

```haskell
eOr2 :: RuleSet Term () -> Rule Term Term
eOr2 value step (TmOr tm1 tm2) = do
  _ <- value tm1
  tm2' <- step tm2
  pure $ TmOr tm1 tm2'
eOr2 _ _ _ =
  Nothing
```

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

```haskell
eOrFalseFalse :: Rule Term Term
eOrFalseFalse _ (TmOr TmFalse TmFalse) =
  Just TmFalse
eOrFalseFalse _ _ =
  Nothing
```

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

```haskell
stepEagerR :: RuleSet Term Term
stepEagerR =
  mkRuleSet [ eOr1, eOr2
            , eOrFalseFalse, eOrFalseTrue, eOrTrueFalse, eOrTrueTrue
            ]
```

```haskell
eOrFalse :: Rule Term Term
eOrFalse _ (TmOr TmFalse tm2) =
  Just tm2
eOrFalse _ _ =
  Nothing
```

$$\frac{}{\textsf{false or } t_2 \longrightarrow t_2}\ (\textsc{E-OrFalse})$$

```haskell
eOrTrue :: Rule Term Term
eOrTrue _ (TmOr TmTrue _) =
  Just TmTrue
eOrTrue _ _ =
  Nothing
```

$$\frac{}{\textsf{true or } t_2 \longrightarrow \textsf{true}}\ (\textrm{E-}\textsc{OrTrue})$$

```haskell
stepLazyR :: RuleSet Term Term
stepLazyR =
  mkRuleSet [eOr1, eOrFalse, eOrTrue]
```

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

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

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

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

> stepEagerR tm

Just (TmOr TmTrue (TmOr TmFalse TmTrue))

> stepEagerR >=> stepEagerR $ tm

Just (TmOr TmTrue TmTrue)

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

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

> stepEagerR tm

Just (TmOr TmTrue (TmOr TmFalse TmTrue))

> stepEagerR >=> stepEagerR $ tm

Just (TmOr TmTrue TmTrue)

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

Just TmTrue

> stepEagerR >=> stepEagerR >=> stepEagerR >=> stepEagerR $ tm
> let tm = TmOr (TmOr TmFalse TmTrue) (TmOr TmFalse TmTrue)
> stepEagerR tm
Just (TmOr TmTrue (TmOr TmFalse TmTrue))
> stepEagerR >=> stepEagerR $ tm
Just (TmOr TmTrue TmTrue)
> stepEagerR >=> stepEagerR >=> stepEagerR $ tm
Just TmTrue
> stepEagerR >=> stepEagerR >=> stepEagerR >=> stepEagerR $ tm
Nothing
```

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

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

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

> stepLazyR tm

Just (TmOr TmTrue (TmOr TmFalse TmTrue))

> stepLazyR >=> stepLazyR $ tm

Just TmTrue
```

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

> stepLazyR tm

Just (TmOr TmTrue (TmOr TmFalse TmTrue))

> stepLazyR >=> stepLazyR $ tm

Just TmTrue

> stepLazyR >=> stepLazyR >=> stepLazyR $ tm
> let tm = TmOr (TmOr TmFalse TmTrue) (TmOr TmFalse TmTrue)
> stepLazyR tm
Just (TmOr TmTrue (TmOr TmFalse TmTrue))
> stepLazyR >=> stepLazyR $ tm
Just TmTrue
> stepLazyR >=> stepLazyR >=> stepLazyR $ tm
Nothing
```

The step rules are deterministic

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

For all terms, if

the term can take a step

then

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

The step rules are deterministic

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

For all terms, if

the term can take a step

then

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

The step rules are deterministic

$$\forall\, t \left(\exists t' \,(t \longrightarrow t') \;\longrightarrow\; \left(\exists t'' \,(t \longrightarrow t'') \;\longrightarrow\; t' = t'' \right)\right)$$

For all terms, if

the term can take a step

then

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

The step rules are deterministic

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

For all terms, if

the term can take a step

then

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

The step rules are deterministic

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

For all terms, if

the term can take a step

then

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

The step rules are deterministic

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

For all terms, if

the term can take a step

then

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

The step rules are deterministic

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

For all terms, if

the term can take a step

then

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

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

```haskell
deterministic :: (Show a, Show b, Eq b)
              => Gen a
              -> [Rule a b]
              -> Property
deterministic ga rs = property $ do
  (a', a1, a2) <- forAll $ do
    a <- ga
    rs1 <- Gen.shuffle rs
    rs2 <- Gen.shuffle rs
    pure (a, mkRuleSet rs1 a, mkRuleSet rs2 a)
  a1 === a2
```

```
boolRulesEagerDeterminstic :: Property
boolRulesEagerDeterminstic =
  deterministic genTerm evalRulesEager

boolRulesLazyDeterminstic :: Property
boolRulesLazyDeterminstic =
  deterministic genTerm evalRulesLazy
```

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

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

  <interactive> passed 100 tests.
True

> check $ boolRulesLazyDeterminstic

  <interactive> passed 100 tests.
True
```

Natural numbers

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

This is a unary number system.

Terms and values (eager)

t	$:=$		
		0	*constant zero*
		succ t	*successor*
		pred t	*predecessor*
v	$:=$		
		0	*zero value*
		succ v	*successor value*

Terms and values (lazy)

$$
\begin{array}{lll}
t & := & \\
 & \mathsf{O} & \textit{constant zero} \\
 & \mathsf{succ}\ t & \textit{successor} \\
 & \mathsf{pred}\ t & \textit{predecessor} \\
v & := & \\
 & \mathsf{O} & \textit{zero value} \\
 & \mathsf{succ}\ t & \textit{successor value}
\end{array}
$$

succ O

$$\mathsf{succ}\,(\mathsf{pred}\,(\mathsf{succ}\;\; O))$$

Small-step semantics (eager)

$$\frac{t_1 \longrightarrow {t_1}'}{\mathsf{succ}\ t_1 \longrightarrow \mathsf{succ}\ {t_1}'} \qquad (\textsc{E-Succ})$$

$$\frac{t_1 \longrightarrow {t_1}'}{\mathsf{pred}\ t_1 \longrightarrow \mathsf{pred}\ {t_1}'} \qquad (\textsc{E-Pred})$$

$$\frac{}{\mathsf{pred}\ \mathsf{O} \longrightarrow \mathsf{O}} \qquad (\textsc{E-PredZero})$$

$$\frac{}{\mathsf{pred}\ (\mathsf{succ}\ v) \longrightarrow v} \qquad (\textsc{E-PredSucc})$$

Small-step semantics (lazy)

$$\frac{t_1 \longrightarrow t_1{}'}{\mathsf{pred}\ t_1 \longrightarrow \mathsf{pred}\ t_1{}'} \quad \text{(E-Pred)}$$

$$\frac{}{\mathsf{pred}\ \mathsf{O} \longrightarrow \mathsf{O}} \quad \text{(E-PredZero)}$$

$$\frac{}{\mathsf{pred}\ \left(\mathsf{succ}\ t\right) \longrightarrow t} \quad \text{(E-PredSucc)}$$

These are both versions of the number 1.

With eager evaluation:

$$\text{succ } O$$

With lazy evaluation:

$$\text{succ}\,(\text{pred}\,(\text{succ } O))$$

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

Under lazy evaluation, we don’t want to take any steps that we don’t need to.

This is fine:

$$\mathsf{succ}\,(\mathsf{pred}\,(\mathsf{succ}\;\; O))$$

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

If we use pred on this:

$$\mathrm{succ}\,(\mathrm{pred}\,(\mathrm{succ}\ O))$$

then the outer succ will be removed and evaluation will continue until we hit a O or end up with another succ on the outside.

If we don't use pred on this:

$$\mathrm{succ}\,(\mathrm{pred}\,(\mathrm{succ}\ O))$$

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

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

```haskell
vZero :: Rule Term ()
vZero _ TmZero =
  Just ()
vZero _ _ =
  Nothing
```

$$\frac{}{\textsf{value O}} \quad (\textsc{V-Zero})$$

```haskell
vSuccEager :: Rule Term ()
vSuccEager value (TmSucc tm) = do
  _ <- value tm
  pure ()
vSuccEager _ _ =
  Nothing
```

$$\frac{}{\textsf{value}\ \ (\textsf{succ v})}\qquad (\textrm{V-S}{\small\textrm{UCC}})$$

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

```haskell
vSuccLazy :: Rule Term ()
vSuccLazy _ (TmSucc tm) =
  pure ()
vSuccLazy _ _ =
  Nothing
```

$$\frac{}{\textsf{value}\ \ (\textsf{succ t})} \quad (\text{V-}\textsc{Succ})$$

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

```haskell
eSucc :: Rule Term Term
eSucc step (TmSucc tm) = do
  tm' <- step tm
  pure $ TmSucc tm'
eSucc _ _ =
  Nothing
```

$$\frac{t_1 \longrightarrow {t_1}'}{\mathsf{succ}\ t_1 \longrightarrow \mathsf{succ}\ {t_1}'} \quad (\text{E-}\textsc{Succ})$$

```haskell
ePred :: Rule Term Term
ePred step (TmPred tm) = do
  tm' <- step tm
  pure $ TmPred tm'
ePred _ _ =
  Nothing
```

$$\frac{t_1 \longrightarrow {t_1}'}{\mathsf{pred}\ t_1 \longrightarrow \mathsf{pred}\ {t_1}'} \quad (\textsc{E-Pred})$$

```haskell
ePredZero :: Rule Term Term
ePredZero _ (TmPred TmZero) =
  Just TmZero
ePredZero _ _ =
  Nothing
```

$$\dfrac{}{\mathsf{pred}\ \mathsf{O} \longrightarrow \mathsf{O}}\ (\textsc{E-PredZero})$$

```haskell
ePredSuccEager :: RuleSet Term () -> Rule Term Term
ePredSuccEager value _ (TmPred (TmSucc tm)) = do
  _ <- value tm
  pure tm
ePredSuccEager _ _ _ =
  Nothing
```

$$\frac{}{\mathsf{pred}\ (\mathsf{succ}\ v) \longrightarrow v}\ (\text{E-}\textsc{PredSucc})$$

```
stepEagerR :: RuleSet Term Term
stepEagerR =
  mkRuleSet [eSucc, ePred, ePredZero, ePredSuccEager valueEagerR]
```

```haskell
ePredSuccLazy :: Rule Term Term
ePredSuccLazy _ (TmPred (TmSucc tm)) =
  Just tm
ePredSuccLazy _ _ =
  Nothing
```

$$\frac{}{\mathsf{pred}\ (\mathsf{succ}\ t) \longrightarrow t}\ (\text{E-}\textsc{PredSucc})$$

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

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

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

> stepEagerR tm

Just (TmSucc TmZero)

> stepEagerR >=> stepEagerR $ tm

Nothing
```

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

> stepLazyR tm
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepLazyR tm
Nothing
```

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

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

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

> stepEagerR $ TmPred tm

Just (TmPred (TmSucc TmZero))

> stepEagerR >=> stepEagerR $ TmPred tm

Just TmZero
```

```
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepEagerR $ TmPred tm
Just (TmPred (TmSucc TmZero))
> stepEagerR >=> stepEagerR $ TmPred tm
Just TmZero
> stepEagerR >=> stepEagerR >=> stepEagerR $ TmPred tm
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepEagerR $ TmPred tm
Just (TmPred (TmSucc TmZero))
> stepEagerR >=> stepEagerR $ TmPred tm
Just TmZero
> stepEagerR >=> stepEagerR >=> stepEagerR $ TmPred tm
Nothing
```

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

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

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

> stepLazyR $ TmPred tm

Just (TmPred (TmSucc TmZero))

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

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

> stepLazyR $ TmPred tm

Just (TmPred (TmSucc TmZero))

> stepLazyR >=> stepLazyR $ TmPred tm

Just TmZero

> stepLazyR >=> stepLazyR >=> stepLazyR $ TmPred tm
> let tm = TmSucc (TmPred (TmSucc TmZero))
> stepLazyR $ TmPred tm
Just (TmPred (TmSucc TmZero))
> stepLazyR >=> stepLazyR $ TmPred tm
Just TmZero
> stepLazyR >=> stepLazyR >=> stepLazyR $ TmPred tm
Nothing
```

Booleans and Natural numbers

We are going to combine a few of these languages.

Terms

t	$:=$		
		false	*constant false*
		true	*constant true*
		t or t	*disjunction*
		O	*constant zero*
		succ t	*successor*
		pred t	*predecessor*
		iszero t	*iszero*
		if t then t else t	*if*

Small-step semantics for iszero (eager)

$$\frac{t \longrightarrow t'}{\text{iszero } t \longrightarrow \text{iszero } t'} \qquad (\textsc{E-IsZero})$$

$$\frac{}{\text{iszero O} \longrightarrow \text{true}} \qquad (\textsc{E-IsZeroZero})$$

$$\frac{}{\text{iszero (succ } v\text{)} \longrightarrow \text{false}} \qquad (\textsc{E-IsZeroSucc})$$

Small-step semantics for iszero (lazy)

$$\frac{t \longrightarrow t'}{\text{iszero } t \longrightarrow \text{iszero } t'} \quad (\text{E-IsZero})$$

$$\frac{}{\text{iszero } 0 \longrightarrow \text{true}} \quad (\text{E-IsZeroZero})$$

$$\frac{}{\text{iszero } (\text{succ } t) \longrightarrow \text{false}} \quad (\text{E-IsZeroSucc})$$

Small-step semantics for if

$$\frac{t_1 \longrightarrow {t_1}'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } {t_1}' \text{ then } t_2 \text{ else } t_3} \qquad (\text{E-IF})$$

$$\frac{}{\text{if true then } t_2 \text{ else } t_3 \longrightarrow t_2} \qquad (\text{E-IFTRUE})$$

$$\frac{}{\text{if false then } t_2 \text{ else } t_3 \longrightarrow t_3} \qquad (\text{E-IFFALSE})$$

This term is stuck:

$$\text{iszero false}$$

as is this one:

$$\text{if 0 then false else true}$$

We can break values down further to try to keep things on-track.

Values

bv	$:=$		
		false	*false value*
		true	*true value*
nv	$:=$		
		O	*zero value*
		succ nv	*successor value*
v	$:=$		
		bv	*boolean value*
		nv	*natural number value*

Small-step semantics

$$\frac{}{\textsf{iszero}\ \ (\textsf{succ}\ \ \mathit{nv}\ ) \longrightarrow \textsf{false}} \qquad (\textsc{E-IsZeroSucc})$$

These terms are still stuck:

iszero false

if 0 then false else true

The finer-grained values have more clearly communicated intent.

The more detailed break down of values may effect *when* a term gets stuck, but not whether a term will get stuck.

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

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

False
```

```
> check boolNatRulesEagerValueOrStep
  <interactive> failed after 4 tests and 2 shrinks.

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

          | Both equal to
          | False

False
```

We want to work out which terms will or won't get stuck without having to evaluate the terms.

Enter *types*.

Types

T :=		
	Bool	type of booleans
	Nat	type of natural numbers

There is a typing relation $\Gamma \vdash t : T$.

This indicates that the term t is *well-typed* and has type T inside the context of Γ.

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

Typing rules (Booleans)

$$\frac{}{\vdash \mathsf{false} : \mathsf{Bool}} \quad \text{(T-False)}$$

$$\frac{}{\vdash \mathsf{true} : \mathsf{Bool}} \quad \text{(T-True)}$$

$$\frac{\vdash t_1 : \mathsf{Bool} \qquad \vdash t_2 : \mathsf{Bool}}{\vdash t_1 \ \mathsf{or}\ t_2 : \mathsf{Bool}} \quad \text{(T-Or)}$$

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

Typing rules (Booleans)

$$\frac{}{\vdash \mathsf{false} : \mathsf{Bool}} \quad \text{(T-False)}$$

$$\frac{}{\vdash \mathsf{true} : \mathsf{Bool}} \quad \text{(T-True)}$$

$$\frac{\vdash t_1 : \mathsf{Bool} \qquad \vdash t_2 : \mathsf{Bool}}{\vdash t_1 \ \mathsf{or}\ t_2 : \mathsf{Bool}} \quad \text{(T-Or)}$$

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

Typing rules (Booleans)

$$\frac{}{\vdash \mathsf{false} : \mathsf{Bool}} \quad (\textsc{T-False})$$

$$\frac{}{\vdash \mathsf{true} : \mathsf{Bool}} \quad (\textsc{T-True})$$

$$\frac{\vdash t_1 : \mathsf{Bool} \qquad \vdash t_2 : \mathsf{Bool}}{\vdash t_1 \ \mathsf{or} \ t_2 : \mathsf{Bool}} \quad (\textsc{T-Or})$$

At this point our type inference is syntax-directed - we can walk through the syntax tree, applying one rule at a time.

Typing rules (Natural numbers)

$$\frac{}{\vdash \mathsf{O} : \mathsf{Nat}} \qquad \text{(T-Zero)}$$

$$\frac{\vdash t : \mathsf{Nat}}{\vdash \mathsf{succ}\ t : \mathsf{Nat}} \qquad \text{(T-Succ)}$$

$$\frac{\vdash t : \mathsf{Nat}}{\vdash \mathsf{pred}\ t : \mathsf{Nat}} \qquad \text{(T-Pred)}$$

Typing rules (both)

$$\frac{\vdash t : \mathsf{Nat}}{\vdash \mathsf{iszero}\ t : \mathsf{Bool}} \qquad (\textsc{T-IsZero})$$

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

Typing rules (both)

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

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

These rule out the stuck terms we saw previously:

$$\mathsf{iszero\ false}$$

and

$$\mathsf{if\ O\ then\ false\ else\ true}$$

Typing rules (both)

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

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

They also rule out terms that are not stuck:

$$\mathsf{if\ true\ then\ O\ else\ true}$$

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

Typing rules (both)

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

$$\frac{\vdash t_1 : \mathsf{Bool} \qquad \vdash t_2 : T \qquad \vdash t_3 : T}{\vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T} \quad (\textsc{T-If})$$

This kind of thing is normally not a big deal.

Typing rules (both)

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

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

When a type system rules out some terms that were not stuck, it is called a *conservative* type system.

For now, each well-typed term will have a unique type.

Later on that will relax, and we'll be more concerned with the *principal type* of a term.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash\ \mathsf{O}\ :\ \mathsf{Nat}}\,\text{T-Zero}}{\vdash\ \mathsf{succ}\ \ \mathsf{O}\ \ :\ \mathsf{Nat}}\,\text{T-Succ}}{\vdash\ \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right)\ :\ \mathsf{Bool}}\,\text{T-IsZero} \quad \dfrac{}{\vdash\ \mathsf{false}\ :\ \mathsf{Bool}}\,\text{T-False} \quad \dfrac{\dfrac{}{\vdash\ \mathsf{true}\ :\ \mathsf{Bool}}\,\text{T-True} \quad \dfrac{}{\vdash\ \mathsf{false}\ :\ \mathsf{Bool}}\,\text{T-False}}{\vdash\ \ \mathsf{true}\ \ \mathsf{or}\ \ \mathsf{false}\ \ :\ \mathsf{Bool}}\,\text{T-Or}}{\vdash\ \mathsf{if}\ \left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \ \mathsf{false}\ \ \mathsf{else}\ \left(\mathsf{true\ or\ false}\right)\ :\ \mathsf{Bool}}\,\text{T-If}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash\ \mathsf{O}\ :\ \mathsf{Nat}}\,\text{T-Zero}}{\vdash\ \mathsf{succ}\ \ \mathsf{O}\ \ :\ \mathsf{Nat}}\,\text{T-Succ}}{\vdash\ \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right)\ :\ \mathsf{Bool}}\,\text{T-IsZero} \qquad \dfrac{}{\vdash\ \mathsf{false}\ :\ \mathsf{Bool}}\,\text{T-False} \qquad \dfrac{\dfrac{}{\vdash\ \mathsf{true}\ :\ \mathsf{Bool}}\,\text{T-True} \quad \dfrac{}{\vdash\ \mathsf{false}\ :\ \mathsf{Bool}}\,\text{T-False}}{\vdash\ \ \mathsf{true}\ \ \mathsf{or}\ \ \mathsf{false}\ \ :\ \mathsf{Bool}}\,\text{T-Or}}{\vdash\ \mathsf{if}\ \left(\mathsf{iszero}\ \ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \ \mathsf{false}\ \ \mathsf{else}\ \left(\mathsf{true\ or\ false}\right)\ :\ \mathsf{Bool}}\,\text{T-If}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\,\text{T-Zero}}{\vdash \mathsf{succ}\ \mathsf{O} : \mathsf{Nat}}\,\text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\,\text{T-IsZero} \qquad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False} \qquad \dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\,\text{T-True} \quad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False}}{\vdash \mathsf{true}\ \mathsf{or}\ \mathsf{false} : \mathsf{Bool}}\,\text{T-Or}}{\vdash \mathsf{if}\left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\left(\mathsf{true}\ \mathsf{or}\ \mathsf{false}\right) : \mathsf{Bool}}\,\text{T-If}$$

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\,\text{T-Zero}}{\vdash \mathsf{succ}\ \mathsf{O} : \mathsf{Nat}}\,\text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\,\text{T-IsZero} \qquad \cfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False} \qquad \cfrac{\cfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\,\text{T-True} \quad \cfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False}}{\vdash \mathsf{true}\ \mathsf{or}\ \mathsf{false} : \mathsf{Bool}}\,\text{T-Or}}{\vdash \mathsf{if}\ \left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \left(\mathsf{true}\ \mathsf{or}\ \mathsf{false}\right) : \mathsf{Bool}}\,\text{T-If}$$

$$
\dfrac{
\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\,\text{T-Zero}}{\vdash \mathsf{succ}\ \mathsf{O} : \mathsf{Nat}}\,\text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\,\text{T-IsZero}
\qquad
\dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False}
\qquad
\dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\,\text{T-True}\quad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False}}{\vdash \mathsf{true}\ \mathsf{or}\ \mathsf{false} : \mathsf{Bool}}\,\text{T-Or}
}{
\vdash \mathsf{if}\left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\left(\mathsf{true}\ \mathsf{or}\ \mathsf{false}\right) : \mathsf{Bool}
}\,\text{T-If}
$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\,\text{T-Zero}}{\vdash \mathsf{succ\ O} : \mathsf{Nat}}\,\text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ\ O}\right) : \mathsf{Bool}}\,\text{T-IsZero} \qquad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False} \qquad \dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\,\text{T-True} \quad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False}}{\vdash \mathsf{true\ or\ false} : \mathsf{Bool}}\,\text{T-Or}}{\vdash \mathsf{if}\left(\mathsf{iszero\ (succ\ O)}\right)\ \mathsf{then\ false\ else}\left(\mathsf{true\ or\ false}\right) : \mathsf{Bool}}\,\text{T-If}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\,\text{T-Zero}}{\vdash \mathsf{succ}\ \mathsf{O} : \mathsf{Nat}}\,\text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\,\text{T-IsZero} \quad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False} \quad \dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\,\text{T-True} \quad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False}}{\vdash \mathsf{true}\ \mathsf{or}\ \mathsf{false} : \mathsf{Bool}}\,\text{T-Or}}{\vdash \mathsf{if}\left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\left(\mathsf{true}\ \mathsf{or}\ \mathsf{false}\right) : \mathsf{Bool}}\,\text{T-If}$$

$$
\dfrac{
\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\,\text{T-Zero}}{\vdash \mathsf{succ}\ \mathsf{O} : \mathsf{Nat}}\,\text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\,\text{T-IsZero}
\quad
\dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False}
\quad
\dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\,\text{T-True} \quad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False}}{\vdash \mathsf{true}\ \mathsf{or}\ \mathsf{false} : \mathsf{Bool}}\,\text{T-Or}
}{\vdash \mathsf{if}\left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\left(\mathsf{true}\ \mathsf{or}\ \mathsf{false}\right) : \mathsf{Bool}}\,\text{T-If}
$$

$$
\dfrac{
\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\ \text{T-Zero}}{\vdash \mathsf{succ}\ \mathsf{O} : \mathsf{Nat}}\ \text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\ \text{T-IsZero}
\qquad
\dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\ \text{T-False}
\qquad
\dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\ \text{T-True} \qquad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\ \text{T-False}}{\vdash \mathsf{true}\ \mathsf{or}\ \mathsf{false} : \mathsf{Bool}}\ \text{T-Or}
}{
\vdash \mathsf{if}\ \left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \left(\mathsf{true}\ \mathsf{or}\ \mathsf{false}\right) : \mathsf{Bool}
}\ \text{T-If}
$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\,\text{T-Zero}}{\vdash \mathsf{succ}\ \mathsf{O} : \mathsf{Nat}}\,\text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\,\text{T-IsZero} \qquad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False} \qquad \dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\,\text{T-True} \quad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False}}{\vdash \mathsf{true}\ \mathsf{or}\ \mathsf{false} : \mathsf{Bool}}\,\text{T-Or}}{\vdash \mathsf{if}\ \left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \left(\mathsf{true\ or\ false}\right) : \mathsf{Bool}}\,\text{T-If}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\,\text{T-Zero}}{\vdash \mathsf{succ}\ \ \mathsf{O} : \mathsf{Nat}}\,\text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\,\text{T-IsZero} \qquad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False} \qquad \dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\,\text{T-True} \quad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False}}{\vdash \mathsf{true}\ \ \mathsf{or}\ \ \mathsf{false} : \mathsf{Bool}}\,\text{T-Or}}{\vdash \mathsf{if}\left(\mathsf{iszero}\ \ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \ \mathsf{false}\ \ \mathsf{else}\left(\mathsf{true\ or\ false}\right) : \mathsf{Bool}}\,\text{T-If}$$

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\;\text{T-Zero}
}{\vdash \mathsf{succ}\ \mathsf{O} : \mathsf{Nat}}\;\text{T-Succ}
}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\;\text{T-IsZero}
\qquad
\dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\;\text{T-False}
\qquad
\dfrac{
\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\;\text{T-True}
\quad
\dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\;\text{T-False}
}{\vdash \mathsf{true}\ \mathsf{or}\ \mathsf{false} : \mathsf{Bool}}\;\text{T-Or}
}{\vdash \mathsf{if}\left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\left(\mathsf{true}\ \mathsf{or}\ \mathsf{false}\right) : \mathsf{Bool}}\;\text{T-If}
$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\,\text{T-Zero}}{\vdash \mathsf{succ}\ \mathsf{O} : \mathsf{Nat}}\,\text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\,\text{T-IsZero} \qquad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False} \qquad \dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\,\text{T-True} \quad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False}}{\vdash \mathsf{true}\ \mathsf{or}\ \mathsf{false} : \mathsf{Bool}}\,\text{T-Or}}{\vdash \mathsf{if}\ \left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \left(\mathsf{true}\ \mathsf{or}\ \mathsf{false}\right) : \mathsf{Bool}}\,\text{T-If}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\;\text{T-Zero}}{\vdash \mathsf{succ}\ \mathsf{O} : \mathsf{Nat}}\;\text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\;\text{T-IsZero} \qquad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\;\text{T-False} \qquad \dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\;\text{T-True} \quad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\;\text{T-False}}{\vdash \mathsf{true}\ \mathsf{or}\ \mathsf{false} : \mathsf{Bool}}\;\text{T-Or}}{\vdash \mathsf{if}\left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\left(\mathsf{true}\ \mathsf{or}\ \mathsf{false}\right) : \mathsf{Bool}}\;\text{T-If}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \text{O} : \text{Nat}}\;\text{T-Zero}}{\vdash \text{succ}\;\;\text{O} : \text{Nat}}\;\text{T-Succ}}{\vdash \text{iszero}\left(\text{succ O}\right) : \text{Bool}}\;\text{T-IsZero} \qquad \dfrac{}{\vdash \text{false} : \text{Bool}}\;\text{T-False} \qquad \dfrac{\dfrac{}{\vdash \text{true} : \text{Bool}}\;\text{T-True} \quad \dfrac{}{\vdash \text{false} : \text{Bool}}\;\text{T-False}}{\vdash \text{true}\;\;\text{or}\;\;\text{false} : \text{Bool}}\;\text{T-Or}}{\vdash \text{if}\left(\text{iszero (succ O)}\right)\text{then false else}\left(\text{true or false}\right) : \text{Bool}}\;\text{T-If}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\;\text{T-Zero}}{\vdash \mathsf{succ}\ \ \mathsf{O} : \mathsf{Nat}}\;\text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\;\text{T-IsZero} \qquad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\;\text{T-False} \qquad \dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\;\text{T-True} \quad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\;\text{T-False}}{\vdash \mathsf{true}\ \ \mathsf{or}\ \ \mathsf{false} : \mathsf{Bool}}\;\text{T-Or}}{\vdash \mathsf{if}\ \left(\mathsf{iszero}\ \ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \left(\mathsf{true}\ \mathsf{or}\ \mathsf{false}\right) : \mathsf{Bool}}\;\text{T-If}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\;\text{T-Zero}}{\vdash \mathsf{succ}\ \mathsf{O} : \mathsf{Nat}}\;\text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\;\text{T-IsZero} \qquad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\;\text{T-False} \qquad \dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\;\text{T-True} \quad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\;\text{T-False}}{\vdash \mathsf{true}\ \mathsf{or}\ \mathsf{false} : \mathsf{Bool}}\;\text{T-Or}}{\vdash \mathsf{if}\ \left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \left(\mathsf{true\ or\ false}\right) : \mathsf{Bool}}\;\text{T-If}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\ \text{T-Zero}}{\vdash \mathsf{succ}\ \mathsf{O} : \mathsf{Nat}}\ \text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\ \text{T-IsZero} \qquad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\ \text{T-False} \qquad \dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\ \text{T-True} \quad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\ \text{T-False}}{\vdash \mathsf{true}\ \mathsf{or}\ \mathsf{false} : \mathsf{Bool}}\ \text{T-Or}}{\vdash \mathsf{if}\left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\left(\mathsf{true}\ \mathsf{or}\ \mathsf{false}\right) : \mathsf{Bool}}\ \text{T-If}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\,\text{T-Zero}}{\vdash \mathsf{succ}\ \mathsf{O} : \mathsf{Nat}}\,\text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\,\text{T-IsZero} \qquad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False} \qquad \dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\,\text{T-True} \quad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False}}{\vdash \mathsf{true}\ \mathsf{or}\ \mathsf{false} : \mathsf{Bool}}\,\text{T-Or}}{\vdash \mathsf{if}\ \left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \left(\mathsf{true\ or\ false}\right) : \mathsf{Bool}}\,\text{T-If}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\,\text{T-Zero}}{\vdash \mathsf{succ}\ \mathsf{O} : \mathsf{Nat}}\,\text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\,\text{T-IsZero} \quad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False} \quad \dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\,\text{T-True} \quad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False}}{\vdash \mathsf{true}\ \mathsf{or}\ \mathsf{false} : \mathsf{Bool}}\,\text{T-Or}}{\vdash \mathsf{if}\left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\left(\mathsf{true\ or\ false}\right) : \mathsf{Bool}}\,\text{T-If}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\,\text{T-Zero}}{\vdash \mathsf{succ}\ \mathsf{O} : \mathsf{Nat}}\,\text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\,\text{T-IsZero} \qquad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False} \qquad \dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\,\text{T-True} \quad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False}}{\vdash \mathsf{true}\ \mathsf{or}\ \mathsf{false} : \mathsf{Bool}}\,\text{T-Or}}{\vdash \mathsf{if}\ \left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \left(\mathsf{true\ or\ false}\right) : \mathsf{Bool}}\,\text{T-If}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\,\text{T-Zero}}{\vdash \mathsf{succ}\ \mathsf{O} : \mathsf{Nat}}\,\text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\,\text{T-IsZero} \qquad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False} \qquad \dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\,\text{T-True} \quad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False}}{\vdash \mathsf{true}\ \mathsf{or}\ \mathsf{false} : \mathsf{Bool}}\,\text{T-Or}}{\vdash \mathsf{if}\ \left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \left(\mathsf{true\ or\ false}\right) : \mathsf{Bool}}\,\text{T-If}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\ \text{T-Zero}}{\vdash \mathsf{succ}\ \mathsf{O} : \mathsf{Nat}}\ \text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\ \text{T-IsZero} \qquad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\ \text{T-False} \qquad \dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\ \text{T-True} \qquad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\ \text{T-False}}{\vdash \mathsf{true}\ \mathsf{or}\ \mathsf{false} : \mathsf{Bool}}\ \text{T-Or}}{\vdash \mathsf{if}\ \left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \left(\mathsf{true}\ \mathsf{or}\ \mathsf{false}\right) : \mathsf{Bool}}\ \text{T-If}$$

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\,\text{T-Zero}
}{\vdash \mathsf{succ}\ \mathsf{O} : \mathsf{Nat}}\,\text{T-Succ}
}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\,\text{T-IsZero}
\qquad
\dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False}
\qquad
\dfrac{
\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\,\text{T-True}
\quad
\dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False}
}{\vdash \mathsf{true}\ \mathsf{or}\ \mathsf{false} : \mathsf{Bool}}\,\text{T-Or}
}{\vdash \mathsf{if}\ \left(\mathsf{iszero}\ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ \left(\mathsf{true\ or\ false}\right) : \mathsf{Bool}}\,\text{T-If}
$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \text{O} : \text{Nat}}\,\text{T-Zero}}{\vdash \text{succ}\ \ \text{O} : \text{Nat}}\,\text{T-Succ}}{\vdash \text{iszero}\left(\text{succ O}\right) : \text{Bool}}\,\text{T-IsZero} \quad \dfrac{}{\vdash \text{false} : \text{Bool}}\,\text{T-False} \quad \dfrac{\dfrac{}{\vdash \text{true} : \text{Bool}}\,\text{T-True} \quad \dfrac{}{\vdash \text{false} : \text{Bool}}\,\text{T-False}}{\vdash \text{true}\ \ \text{or}\ \ \text{false} : \text{Bool}}\,\text{T-Or}}{\vdash \text{if}\ \left(\text{iszero}\ \ (\text{succ O})\right)\ \text{then}\ \ \text{false}\ \ \text{else}\ \left(\text{true or false}\right) : \text{Bool}}\,\text{T-If}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\,\text{T-Zero}}{\vdash \mathsf{succ}\ \ \mathsf{O} : \mathsf{Nat}}\,\text{T-Succ}}{\vdash \mathsf{iszero}\left(\mathsf{succ}\ \mathsf{O}\right) : \mathsf{Bool}}\,\text{T-IsZero} \qquad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False} \qquad \dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\,\text{T-True} \qquad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\,\text{T-False}}{\vdash \mathsf{true}\ \ \mathsf{or}\ \ \mathsf{false} : \mathsf{Bool}}\,\text{T-Or}}{\vdash \mathsf{if}\ \left(\mathsf{iszero}\ \ (\mathsf{succ}\ \mathsf{O})\right)\ \mathsf{then}\ \ \mathsf{false}\ \ \mathsf{else}\ \left(\mathsf{true\ or\ false}\right) : \mathsf{Bool}}\,\text{T-If}$$

$$
\dfrac{
\dfrac{\dfrac{\dfrac{}{\vdash \mathsf{O} : \mathsf{Nat}}\ \text{T-Zero}}{\vdash \mathsf{succ}\ \ \mathsf{O} : \mathsf{Nat}}\ \text{T-Succ}}{\vdash \mathsf{iszero}\left(\ \mathsf{succ}\ \mathsf{O}\ \right) : \mathsf{Bool}}\ \text{T-IsZero}
\qquad
\dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\ \text{T-False}
\qquad
\dfrac{\dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\ \text{T-True} \qquad \dfrac{}{\vdash \mathsf{false} : \mathsf{Bool}}\ \text{T-False}}{\vdash \mathsf{true}\ \ \mathsf{or}\ \ \mathsf{false} : \mathsf{Bool}}\ \text{T-Or}
}{
\vdash \mathsf{if}\ \left(\ \mathsf{iszero}\ \ (\mathsf{succ}\ \mathsf{O})\ \right)\ \mathsf{then}\ \ \mathsf{false}\ \ \mathsf{else}\ \left(\ \mathsf{true}\ \mathsf{or}\ \mathsf{false}\ \right) : \mathsf{Bool}
}\ \text{T-If}
$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \text{O} : \text{Nat}}\,\text{T-Zero}}{\vdash \text{succ O} : \text{Nat}}\,\text{T-Succ}}{\vdash \text{iszero}\left(\text{succ O}\right) : \text{Bool}}\,\text{T-IsZero} \qquad \dfrac{}{\vdash \text{false} : \text{Bool}}\,\text{T-False} \qquad \dfrac{\dfrac{}{\vdash \text{true} : \text{Bool}}\,\text{T-True} \quad \dfrac{}{\vdash \text{false} : \text{Bool}}\,\text{T-False}}{\vdash \text{true or false} : \text{Bool}}\,\text{T-Or}}{\vdash \text{if}\left(\text{iszero (succ O)}\right)\text{ then false else }\left(\text{true or false}\right) : \text{Bool}}\,\text{T-If}$$

```haskell
data Type =
    TyBool
  | TyNat
  deriving (Eq, Ord, Show)
```

```haskell
checkIsZero :: Rule (Term, Type) ()
checkIsZero step (TmIsZero tm, TyBool) =
  step (tm, TyNat)
checkIsZero _ _ =
  Nothing
```

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

```haskell
checkIf :: Rule (Term, Type) ()
checkIf step (TmIf tm1 tm2 tm3, ty) = do
  step (tm1, TyBool)
  step (tm2, ty)
  step (tm3, ty)
checkIf _ _ =
  Nothing
```

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

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

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

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

```haskell
expect :: Eq ty => RuleSet tm ty -> tm -> ty -> Maybe ()
expect step tm ty = do
  ty' <- step tm
  if (ty == ty')
  then Just ()
  else Nothing

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

```haskell
inferIsZero :: Rule Term Type
inferIsZero step (TmIsZero tm) = do
  expect step tm TyNat
  pure TyBool
inferIsZero _ _ =
  Nothing
```

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

```haskell
inferIf :: Rule Term Type
inferIf step (TmIf tm1 tm2 tm3) = do
  expect step tm1 TyBool
  expectEq step tm2 tm3
inferIf _ _ =
  Nothing
```

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

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

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

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

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

genWellTypedTerm :: Gen Term
genWellTypedTerm =
  genType >>= genTypedTerm
```

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

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

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

```
> check $ checkTypeCorrect

  <interactive> passed 100 tests.
True

> check $ inferCorrect
> check $ checkTypeCorrect

  <interactive> passed 100 tests.
True

> check $ inferCorrect

  <interactive> passed 100 tests.
True
```

There are two main properties which relate the type system of a language and the small-step semantics of a language.

Progress

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

For all well-typed terms,

the term is either a value or

is able to step .

Progress

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

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

Progress

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

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

Progress

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

For all well-typed terms,

the term is either a value or

is able to step .

Progress

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

For all well-typed terms,

the term is either a value or

is able to step .

Progress

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

For all well-typed terms,

the term is either a value or

is able to step.

Preservation

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

For all well-typed terms ,

if the term can take a step

the type is unchanged .

Preservation

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

For all well-typed terms ,

if the term can take a step

the type is unchanged .

Preservation

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

For all well-typed terms,

if the term can take a step

the type is unchanged .

Preservation

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

For all well-typed terms ,

if the term can take a step

the type is unchanged .

Preservation

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

For all well-typed terms ,

if the term can take a step

the type is unchanged .

Preservation

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

For all well-typed terms ,

if the term can take a step

the type is unchanged .

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.

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

- Progress means that well-typed terms are either values or can take a step.
- Values are not stuck, so if we are at a value we are done.

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

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

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

preservation :: RuleSet Term Term -> Property
preservation step = property $ do
  (ty, tm) <- forAll $ do
    ty' <- genType
    tm' <- Gen.filter (isJust . step) (genTypedTerm ty')
    pure (ty', tm')
  Just ty === (step tm >>= infer)
```

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

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

  <interactive> passed 100 tests.
True

> check $ preservation stepEagerR

  <interactive> passed 100 tests.
True
```

Lambda Calculus

This is where we step things up a notch.

Terms and values

t	$:=$	$\ldots$	
		x	*variable*
		$\lambda\ x.t$	*abstraction*
		$t\ \ t$	*function application*
v	$:=$	$\ldots$	
		$\lambda\ x.t$	*abstraction*

Lambda anatomy

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

Lambda anatomy

$$\lambda \; x \; . \; x + 2$$

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

Lambda anatomy

$$\lambda\, x\ .\ \boxed{x} + 2$$

The x to the right of the . is a variable.

Let us look at some terms.

The term

$$x$$

is meaningless.

The term

$$x + 2$$

is also meaningless.

The term

$$\lambda\, x\ .\ x + 2$$

is an anonymous equivalent to

$$f(x) = x + 2$$

The term

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

is equivalent to

$$f(1)$$

when f is defined as

$$f(x) = x + 2$$

like before.

In ordinary maths, we evaluate

$$f(1)$$

by taking

$$f(x) = x + 2$$

and replacing the occurrences of x with 1 to get

$$f(1) = 1 + 2$$

The notation for that kind of replacement is

$$[x \mapsto 1]\, f$$

We would like to see something similar happening in our evaluation rules:

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

We need to be careful with substitution.

When we evaluate

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

we have

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

We want

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

to become

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

instead of

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

In order to do that, we need to know the *free variables* of

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

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

$$\lambda\, x\ .\ \boxed{x} + 1$$

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.

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

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

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

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

$$((\lambda\, x\, .\; x + 1)\;\; (\, x\, + 1))$$

In

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

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

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

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

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

Free variable rules

$$\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}$$

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

Free variable rules

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\ FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\ \\ FV(\langle int \rangle) &= \emptyset \\ FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2) \end{aligned}$$

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

Free variable rules

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}
$$

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

Free variable rules

$$\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}$$

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

Free variable rules

$$\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}$$

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

Free variable rules

$$\begin{array}{lcl} FV(x) & = & \{x\} \\ FV(\lambda\ x.t_1) & = & FV(t_1) \setminus \{x\} \\ FV(t_1\ \ t_2) & = & FV(t_1) \cup FV(t_2) \\ \\ FV(\langle int \rangle) & = & \emptyset \\ FV(t_1 + t_2) & = & FV(t_1) \cup FV(t_2) \end{array}$$

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

Free variable rules

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}
$$

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

Free variable rules

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}
$$

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

Free variable rules

$$\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}$$

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

Free variable rules

$$\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}$$

$$\begin{aligned}
FV((\lambda\ x.x+1)\ \ (x+1)) &= FV(\lambda\ x.x+1) \cup\ FV(x+1) \\
&= (FV(x+1) \setminus \{x\}) \cup\ FV(x+1)
\end{aligned}$$

Free variable rules

$$\begin{array}{lcl}
FV(x) & = & \{x\} \\
FV(\lambda\ x.t_1) & = & FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) & = & FV(t_1) \cup FV(t_2) \\
 & & \\
FV(\langle int \rangle) & = & \emptyset \\
FV(t_1 + t_2) & = & FV(t_1) \cup FV(t_2)
\end{array}$$

$$\begin{array}{lcl}
FV((\lambda\ x.x+1)\ \ (x+1)) & = & FV(\lambda\ x.x+1)\ \cup\ FV(x+1) \\
 & = & (FV(x+1) \setminus \{x\})\ \cup\ FV(x+1)
\end{array}$$

Free variable rules

$$\begin{array}{lcl}
FV(x) & = & \{x\} \\
FV(\lambda\ x.t_1) & = & FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) & = & FV(t_1) \cup FV(t_2) \\
 & & \\
FV(\langle int \rangle) & = & \emptyset \\
FV(t_1 + t_2) & = & FV(t_1) \cup FV(t_2)
\end{array}$$

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

Free variable rules

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}
$$

$$
\begin{aligned}
FV((\lambda\ x.x+1)\ \ (x+1)) &= (FV(x+1) \setminus \{x\})\ \cup\ FV(x+1) \\
&= FV(x+1)
\end{aligned}
$$

Free variable rules

$$\begin{array}{lcl} FV(x) & = & \{x\} \\ FV(\lambda\ x.t_1) & = & FV(t_1) \setminus \{x\} \\ FV(t_1\ \ t_2) & = & FV(t_1) \cup FV(t_2) \\ \\ FV(\langle int \rangle) & = & \emptyset \\ FV(t_1 + t_2) & = & FV(t_1) \cup FV(t_2) \end{array}$$

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

Free variable rules

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}
$$

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

Free variable rules

$$\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}$$

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

Free variable rules

$$\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}$$

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

Free variable rules

$$\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}$$

$$\begin{aligned}
FV((\lambda\ x.x + 1)\ \ (x + 1)) &= FV(x + 1) \\
&= FV(x)\ \cup\ FV(1)
\end{aligned}$$

Free variable rules

$$\begin{array}{lcl}
FV(x) & = & \{x\} \\
FV(\lambda\ x.t_1) & = & FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) & = & FV(t_1) \cup FV(t_2) \\
 & & \\
FV(\langle int \rangle) & = & \emptyset \\
FV(t_1 + t_2) & = & FV(t_1) \cup FV(t_2)
\end{array}$$

$$\begin{array}{lcl}
FV((\lambda\ x.x + 1)\ \ (x + 1)) & = & FV(x + 1) \\
 & = & FV(x)\ \cup\ FV(1)
\end{array}$$

Free variable rules

$$\begin{aligned}
FV(x) &= \{x\}\\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\}\\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2)\\
\\
FV(\langle int \rangle) &= \emptyset\\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}$$

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

Free variable rules

$$\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}$$

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

Free variable rules

$$\begin{array}{lcl}
FV(x) & = & \{x\} \\
FV(\lambda\ x.t_1) & = & FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) & = & FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) & = & \emptyset \\
FV(t_1 + t_2) & = & FV(t_1) \cup FV(t_2)
\end{array}$$

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

Free variable rules

$$\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}$$

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

Free variable rules

$$\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}$$

$$\begin{aligned}
FV((\lambda\ x.x + 1)\ \ (x + 1)) &= FV(x) \cup\ FV(1) \\
&= \{x\} \cup\ FV(1)
\end{aligned}$$

Free variable rules

$$
\begin{array}{lcl}
FV(x) & = & \{x\} \\
FV(\lambda\ x.t_1) & = & FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) & = & FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) & = & \emptyset \\
FV(t_1 + t_2) & = & FV(t_1) \cup FV(t_2)
\end{array}
$$

$$
\begin{array}{lcl}
FV((\lambda\ x.x + 1)\ \ (x + 1)) & = & FV(x)\ \cup\ FV(1) \\
 & = & \{x\}\ \cup\ FV(1)
\end{array}
$$

Free variable rules

$$\begin{array}{lcl} FV(x) & = & \{x\} \\ FV(\lambda\ x.t_1) & = & FV(t_1) \setminus \{x\} \\ FV(t_1\ \ t_2) & = & FV(t_1) \cup FV(t_2) \\ \\ FV(\langle int \rangle) & = & \emptyset \\ FV(t_1 + t_2) & = & FV(t_1) \cup FV(t_2) \end{array}$$

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

Free variable rules

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}
$$

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

Free variable rules

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}
$$

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

Free variable rules

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}
$$

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

Free variable rules

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}
$$

$$
\begin{aligned}
FV((\lambda\ x.x + 1)\ \ (x + 1)) &= \{x\} \cup FV(1) \\
&= \{x\} \cup \emptyset
\end{aligned}
$$

Free variable rules

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda\ x.t_1) &= FV(t_1) \setminus \{x\} \\
FV(t_1\ \ t_2) &= FV(t_1) \cup FV(t_2) \\
\\
FV(\langle int \rangle) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2)
\end{aligned}
$$

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

Free variable rules

$$\begin{array}{lcl} FV(x) & = & \{x\} \\ FV(\lambda\ x.t_1) & = & FV(t_1) \setminus \{x\} \\ FV(t_1\ \ t_2) & = & FV(t_1) \cup FV(t_2) \\ \\ FV(\langle int \rangle) & = & \emptyset \\ FV(t_1 + t_2) & = & FV(t_1) \cup FV(t_2) \end{array}$$

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

We are now equipped to deal with substitution.

Substitution rules

$$\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.\,([x \mapsto s]\, t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\, t_1)\ \ ([x \mapsto s]\, t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\, t_1) + ([x \mapsto s]\, t_2) &
\end{array}$$

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

Substitution rules

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.([x \mapsto s]\, t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\, t_1)\ \ ([x \mapsto s]\, t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\, t_1) + ([x \mapsto s]\, t_2) &
\end{array}
$$

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

Substitution rules

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.\,([x \mapsto s]\,t_1) & \text{if } y \neq x \land y \notin FV(s) \\
[x \mapsto s]\ (t_1\ t_2) & = & ([x \mapsto s]\,t_1)\ \ ([x \mapsto s]\,t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\,t_1) + ([x \mapsto s]\,t_2) &
\end{array}
$$

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

Substitution rules

$$\begin{array}{llll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.([x \mapsto s]\, t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ t_2) & = & ([x \mapsto s]\, t_1)\ \ ([x \mapsto s]\, t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\, t_1) + ([x \mapsto s]\, t_2) &
\end{array}$$

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

Substitution rules

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.\,([x \mapsto s]\,t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ t_2) & = & ([x \mapsto s]\,t_1)\ \ ([x \mapsto s]\,t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\,t_1) + ([x \mapsto s]\,t_2) &
\end{array}
$$

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

Substitution rules

$$\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.\,([x \mapsto s]\, t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\, t_1)\ \ ([x \mapsto s]\, t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\, t_1) + ([x \mapsto s]\, t_2) &
\end{array}$$

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

Substitution rules

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.\,([x \mapsto s]\,t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\,t_1)\ \ ([x \mapsto s]\,t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\,t_1) + ([x \mapsto s]\,t_2) &
\end{array}
$$

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

Substitution rules

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.\,([x \mapsto s]\,t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\,t_1)\ \ ([x \mapsto s]\,t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\,t_1) + ([x \mapsto s]\,t_2) &
\end{array}
$$

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

Substitution rules

$$\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.\,([x \mapsto s]\,t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\,t_1)\ \ ([x \mapsto s]\,t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\,t_1) + ([x \mapsto s]\,t_2) &
\end{array}$$

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

Substitution rules

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.\,([x \mapsto s]\,t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\,t_1)\ \ ([x \mapsto s]\,t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\,t_1) + ([x \mapsto s]\,t_2) &
\end{array}
$$

$$
\begin{aligned}
[x \mapsto 3]\ ((\lambda\ x.x+1)\ \ (x+1)) &= \Big([x \mapsto 3]\Big(\lambda\ x.x+1\Big)\Big)\quad\Big([x \mapsto 3]\Big(\ x+1\ \Big)\Big) \\
&= \Big(\lambda\ x.x+1\Big)\quad\Big([x \mapsto 3]\Big(\ x+1\ \Big)\Big)
\end{aligned}
$$

Substitution rules

$$\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.([x \mapsto s]\, t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ t_2) & = & ([x \mapsto s]\, t_1)\ \ ([x \mapsto s]\, t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\, t_1) + ([x \mapsto s]\, t_2) &
\end{array}$$

$$\begin{aligned}
[x \mapsto 3]\ ((\lambda\ x.x+1)\ \ (x+1)) &= \Big([x \mapsto 3]\Big(\ \lambda\ x.x+1\ \Big)\Big)\quad \Big([x \mapsto 3]\Big(\ x+1\ \Big)\Big) \\
&= \Big(\ \lambda\ x.x+1\ \Big)\quad \Big([x \mapsto 3]\Big(\ x+1\ \Big)\Big)
\end{aligned}$$

Substitution rules

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.\,([x \mapsto s]\, t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\, t_1)\ \ ([x \mapsto s]\, t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\, t_1) + ([x \mapsto s]\, t_2) &
\end{array}
$$

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

Substitution rules

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x. t_1) & = & \lambda x. t_1 & \\
[x \mapsto s]\ (\lambda y. t_1) & = & \lambda y. ([x \mapsto s]\, t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\, t_1)\ \ ([x \mapsto s]\, t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\, t_1) + ([x \mapsto s]\, t_2) &
\end{array}
$$

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

Substitution rules

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.\,([x \mapsto s]\,t_1) & \text{if } y \neq x \land y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\,t_1)\ \ ([x \mapsto s]\,t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\,t_1) + ([x \mapsto s]\,t_2) &
\end{array}
$$

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

Substitution rules

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.([x \mapsto s]\, t_1) & \text{if } y \neq x \land y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\, t_1)\ \ ([x \mapsto s]\, t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\, t_1) + ([x \mapsto s]\, t_2) &
\end{array}
$$

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

Substitution rules

$$\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x. t_1) & = & \lambda x. t_1 & \\
[x \mapsto s]\ (\lambda y. t_1) & = & \lambda y. ([x \mapsto s]\, t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\, t_1)\ \ ([x \mapsto s]\, t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\, t_1) + ([x \mapsto s]\, t_2) &
\end{array}$$

$$\begin{aligned}
[x \mapsto 3]\ ((\lambda\ x.x + 1)\ \ (x + 1)) &= \Big(\ \lambda\ x.x + 1\ \Big)\ \ \Big([x \mapsto 3]\Big(x + 1\Big)\Big) \\
&= \Big(\ \lambda\ x.x + 1\ \Big)\ \ \Big([x \mapsto 3]\ x\ \ + [x \mapsto 3]\ 1\ \Big)
\end{aligned}$$

Substitution rules

$$\begin{array}{lll}
[x \mapsto s]\ x & = s & \\
[x \mapsto s]\ y & = y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = \lambda y.\,([x \mapsto s]\,t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = ([x \mapsto s]\,t_1)\ \ ([x \mapsto s]\,t_2) & \\
 & & \\
[x \mapsto s]\ \langle int \rangle & = \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = ([x \mapsto s]\,t_1) + ([x \mapsto s]\,t_2) &
\end{array}$$

$$\begin{aligned}
[x \mapsto 3]\ ((\lambda\ x.x+1)\ \ (x+1)) &= \Big(\ \lambda\ x.x+1\ \Big)\ \ \Big([x \mapsto 3]\Big(\ x+1\ \Big)\Big) \\
&= \Big(\ \lambda\ x.x+1\ \Big)\ \ \Big([x \mapsto 3]\ x\ \ + [x \mapsto 3]\ 1\ \Big)
\end{aligned}$$

Substitution rules

$$\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.\,([x \mapsto s]\,t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\,t_1)\ \ ([x \mapsto s]\,t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\,t_1) + ([x \mapsto s]\,t_2) &
\end{array}$$

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

Substitution rules

$$\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.([x \mapsto s]\, t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\, t_1)\ \ ([x \mapsto s]\, t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\, t_1) + ([x \mapsto s]\, t_2) &
\end{array}$$

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

Substitution rules

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.\,([x \mapsto s]\,t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\,t_1)\ \ ([x \mapsto s]\,t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\,t_1) + ([x \mapsto s]\,t_2) &
\end{array}
$$

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

Substitution rules

$$
\begin{array}{lll}
[x \mapsto s]\ x & = s & \\
[x \mapsto s]\ y & = y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = \lambda y.\,([x \mapsto s]\, t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = ([x \mapsto s]\, t_1)\ \ ([x \mapsto s]\, t_2) & \\
 & & \\
[x \mapsto s]\ \langle int \rangle & = \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = ([x \mapsto s]\, t_1) + ([x \mapsto s]\, t_2) &
\end{array}
$$

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

Substitution rules

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.([x \mapsto s]\, t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\, t_1)\ \ ([x \mapsto s]\, t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\, t_1) + ([x \mapsto s]\, t_2) &
\end{array}
$$

$$
\begin{aligned}
[x \mapsto 3]\ ((\lambda\ x.x+1)\ \ (x+1)) &= \Big(\lambda\ x.x+1\Big)\ \ \Big([x \mapsto 3]\ x + [x \mapsto 3]\ 1\Big) \\
&= \Big(\lambda\ x.x+1\Big)\ \ \Big(3 + [x \mapsto 3]\ 1\Big)
\end{aligned}
$$

Substitution rules

$$\begin{array}{lll}
[x \mapsto s]\ x & = s & \\
[x \mapsto s]\ y & = y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = \lambda y.\,([x \mapsto s]\,t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = ([x \mapsto s]\,t_1)\ \ ([x \mapsto s]\,t_2) & \\
 & & \\
[x \mapsto s]\ \langle int \rangle & = \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = ([x \mapsto s]\,t_1) + ([x \mapsto s]\,t_2) &
\end{array}$$

$$\begin{aligned}
[x \mapsto 3]\ ((\lambda\ x.x+1)\ \ (x+1)) &= \Big(\ \lambda\ x.x+1\ \Big)\ \ \Big([x \mapsto 3]\ x\ + [x \mapsto 3]\ 1\ \Big) \\
&= \Big(\ \lambda\ x.x+1\ \Big)\ \ \Big(\ 3\ + [x \mapsto 3]\ 1\ \Big)
\end{aligned}$$

Substitution rules

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.\,([x \mapsto s]\,t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\,t_1)\ \ ([x \mapsto s]\,t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\,t_1) + ([x \mapsto s]\,t_2) &
\end{array}
$$

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

Substitution rules

$$\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.([x \mapsto s]\, t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\, t_1)\ \ ([x \mapsto s]\, t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\, t_1) + ([x \mapsto s]\, t_2) &
\end{array}$$

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

Substitution rules

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.([x \mapsto s]\,t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ t_2) & = & ([x \mapsto s]\,t_1)\ \ ([x \mapsto s]\,t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\,t_1) + ([x \mapsto s]\,t_2) &
\end{array}
$$

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

Substitution rules

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.([x \mapsto s]\, t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\, t_1)\ \ ([x \mapsto s]\, t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\, t_1) + ([x \mapsto s]\, t_2) &
\end{array}
$$

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

Substitution rules

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.\,([x \mapsto s]\, t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\, t_1)\ \ ([x \mapsto s]\, t_2) & \\
 & & & \\
[x \mapsto s]\ \boxed{\langle int \rangle} & = & \boxed{\langle int \rangle} & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\, t_1) + ([x \mapsto s]\, t_2) &
\end{array}
$$

$$
\begin{aligned}
[x \mapsto 3]\ ((\lambda\ x.x + 1)\ \ (x + 1)) &= \Big(\ \lambda\ x.x + 1\ \Big)\ \ \Big(\ 3\ + [x \mapsto 3]\ \boxed{1}\Big) \\
&= \Big(\ \lambda\ x.x + 1\ \Big)\ \ \Big(\ 3\ + \boxed{1}\Big)
\end{aligned}
$$

Substitution rules

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.\,([x \mapsto s]\,t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\,t_1)\ \ ([x \mapsto s]\,t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\,t_1) + ([x \mapsto s]\,t_2) &
\end{array}
$$

$$
\begin{aligned}
[x \mapsto 3]\ ((\lambda\ x.x+1)\ \ (x+1)) &= \Big(\lambda\ x.x+1\Big)\ \ \Big(3\ +[x \mapsto 3]\ 1\Big) \\
&= \Big(\lambda\ x.x+1\Big)\ \ \Big(3\ +\ 1\Big)
\end{aligned}
$$

Substitution rules

$$
\begin{array}{lcll}
[x \mapsto s]\ x & = & s & \\
[x \mapsto s]\ y & = & y & \text{if } y \neq x \\
[x \mapsto s]\ (\lambda x.t_1) & = & \lambda x.t_1 & \\
[x \mapsto s]\ (\lambda y.t_1) & = & \lambda y.([x \mapsto s]\, t_1) & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s]\ (t_1\ \ t_2) & = & ([x \mapsto s]\, t_1)\ \ ([x \mapsto s]\, t_2) & \\
 & & & \\
[x \mapsto s]\ \langle int \rangle & = & \langle int \rangle & \\
[x \mapsto s]\ (t_1 + t_2) & = & ([x \mapsto s]\, t_1) + ([x \mapsto s]\, t_2) &
\end{array}
$$

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

Small-step semantics (eager)

$$\frac{t_1 \longrightarrow {t_1}'}{t_1 \; t_2 \longrightarrow {t_1}' \; t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow {t_2}'}{v_1 \; t_2 \longrightarrow v_1 \; {t_2}'} \qquad \text{(E-App2)}$$

$$\frac{}{(\lambda\ x.t_{12}) \; v_2 \longrightarrow \left[x \mapsto v_2\right] \; t_{12}} \qquad \text{(E-AppLam)}$$

Small-step semantics (lazy)

$$\frac{t_1 \longrightarrow t_1{}'}{t_1\ t_2 \longrightarrow t_1{}'\ t_2} \qquad (\textsc{E-App1})$$

$$\frac{}{(\lambda\ x.t_{12})\ \boxed{t_2} \longrightarrow \left[x \mapsto \boxed{t_2}\right]\ t_{12}} \qquad (\textsc{E-AppLam})$$

```haskell
data Term a =
    TmVar a
  | TmApp (Term a) (Term a)
  | TmLam String (Scope () Term a)
  ...
  deriving (Eq, Ord, Show, Functor, Foldable, Traversable)
```

```haskell
instance Monad Term where
  return = TmVar

  TmVar x >>= f = f x
  TmApp tm1 tm2 >>= f = TmApp (tm1 >>= f) (tm2 >>= f)
  TmLam v s >>= f = TmLam v (s >>>= f)
  ...
```

```
data Scope b f a = ...
```

```haskell
data Scope b f a = ...

abstract1     :: (Monad f, Eq a) => a -> f a -> Scope () f a
```

```haskell
data Scope b f a = ...
abstract1    :: (Monad f, Eq a) => a -> f a -> Scope () f a
instantiate1 :: Monad f => f a -> Scope n f a -> f a
```

```
> let fn = TmAdd (TmVar "x") (TmInt 1)
> let fn = TmAdd (TmVar "x") (TmInt 1)

> fn
> let fn = TmAdd (TmVar "x") (TmInt 1)

> fn

TmAdd (TmVar "x") (TmInt 1)
```

```
> let fn = TmAdd (TmVar "x") (TmInt 1)

> fn

TmAdd (TmVar "x") (TmInt 1)

> toList fn
> let fn = TmAdd (TmVar "x") (TmInt 1)

> fn

TmAdd (TmVar "x") (TmInt 1)

> toList fn

["x"]
```

```
> let fn = TmAdd (TmVar "x") (TmInt 1)

> fn

TmAdd (TmVar "x") (TmInt 1)

> toList fn

["x"]

> let s = abstract1 "x" fn
> let fn = TmAdd (TmVar "x") (TmInt 1)

> fn

TmAdd (TmVar "x") (TmInt 1)

> toList fn

["x"]

> let s = abstract1 "x" fn

> s
> let fn = TmAdd (TmVar "x") (TmInt 1)

> fn

TmAdd (TmVar "x") (TmInt 1)

> toList fn

["x"]

> let s = abstract1 "x" fn

> s

Scope (TmAdd (TmVar (B ())) (TmInt 1))
```

```
> let fn = TmAdd (TmVar "x") (TmInt 1)

> fn

TmAdd (TmVar "x") (TmInt 1)

> toList fn

["x"]

> let s = abstract1 "x" fn

> s

Scope (TmAdd (TmVar (B ())) (TmInt 1))

> toList s
> let fn = TmAdd (TmVar "x") (TmInt 1)

> fn

TmAdd (TmVar "x") (TmInt 1)

> toList fn

["x"]

> let s = abstract1 "x" fn

> s

Scope (TmAdd (TmVar (B ())) (TmInt 1))

> toList s

[]
```

```
> let fn = TmAdd (TmVar "x") (TmInt 1)

> fn

TmAdd (TmVar "x") (TmInt 1)

> toList fn

["x"]

> let s = abstract1 "x" fn

> s

Scope (TmAdd (TmVar (B ())) (TmInt 1))

> toList s

[]

> instantiate1 (TmInt 3) s
> let fn = TmAdd (TmVar "x") (TmInt 1)

> fn

TmAdd (TmVar "x") (TmInt 1)

> toList fn

["x"]

> let s = abstract1 "x" fn

> s

Scope (TmAdd (TmVar (B ())) (TmInt 1))

> toList s

[]

> instantiate1 (TmInt 3) s

TmAdd (TmInt 3) (TmInt 1)
```

```
> let s' = abstract1 "y" fn
> let s' = abstract1 "y" fn

> s'
> let s' = abstract1 "y" fn

> s'

Scope (TmAdd (TmVar (F (TmVar "x"))) (TmInt 1))
```

```
> let s' = abstract1 "y" fn

> s'

Scope (TmAdd (TmVar (F (TmVar "x"))) (TmInt 1))

> toList s'
> let s' = abstract1 "y" fn

> s'

Scope (TmAdd (TmVar (F (TmVar "x"))) (TmInt 1))

> toList s'

["x"]
```

```
> let s' = abstract1 "y" fn

> s'

Scope (TmAdd (TmVar (F (TmVar "x"))) (TmInt 1))

> toList s'

["x"]

> instantiate1 (TmInt 3) s'
> let s' = abstract1 "y" fn

> s'

Scope (TmAdd (TmVar (F (TmVar "x"))) (TmInt 1))

> toList s'

["x"]

> instantiate1 (TmInt 3) s'

TmAdd (TmVar "x") (TmInt 1)
```

```haskell
lam :: String -> Term String -> Term String
lam v tm = TmLam v (abstract1 v tm)
lam :: String -> Term String -> Term String
lam v tm = TmLam v (abstract1 v tm)
```

```
> let fn = lam "x" $ TmAdd (TmVar "x") (TmInt 1)
```

```haskell
lam :: String -> Term String -> Term String
lam v tm = TmLam v (abstract1 v tm)
```

```
> let fn = lam "x" $ TmAdd (TmVar "x") (TmInt 1)

> fn
```

```haskell
lam :: String -> Term String -> Term String
lam v tm = TmLam v (abstract1 v tm)
```

```
> let fn = lam "x" $ TmAdd (TmVar "x") (TmInt 1)
```

```
> fn
```

```
TmLam "x"
  (Scope (TmAdd (TmVar (B ())) (TmInt 1)))
```

```haskell
lam :: String -> Term String -> Term String
lam v tm = TmLam v (abstract1 v tm)
```

```
> let fn = lam "x" $ TmAdd (TmVar "x") (TmInt 1)
```

```
> fn
```

```
TmLam "x"
  (Scope (TmAdd (TmVar (B ())) (TmInt 1)))
```

```
> let tm = TmApp fn (TmAdd (TmVar "x") (TmInt 1)
```

```haskell
lam :: String -> Term String -> Term String
lam v tm = TmLam v (abstract1 v tm)
```

```
> let fn = lam "x" $ TmAdd (TmVar "x") (TmInt 1)
```

```
> fn
```

```
TmLam "x"
  (Scope (TmAdd (TmVar (B ())) (TmInt 1)))
```

```
> let tm = TmApp fn (TmAdd (TmVar "x") (TmInt 1)
```

```
> tm
```

```haskell
lam :: String -> Term String -> Term String
lam v tm = TmLam v (abstract1 v tm)
```

```
> let fn = lam "x" $ TmAdd (TmVar "x") (TmInt 1)

> fn

TmLam "x"
  (Scope (TmAdd (TmVar (B ())) (TmInt 1)))

> let tm = TmApp fn (TmAdd (TmVar "x") (TmInt 1)

> tm

TmApp
  (TmLam "x" (Scope (TmAdd (TmVar (B ())) (TmInt 1))))
  (TmAdd (TmVar "x") (TmInt 1))
```

```haskell
lam :: String -> Term String -> Term String
lam v tm = TmLam v (abstract1 v tm)

> let fn = lam "x" $ TmAdd (TmVar "x") (TmInt 1)

> fn

TmLam "x"
  (Scope (TmAdd (TmVar (B ())) (TmInt 1)))

> let tm = TmApp fn (TmAdd (TmVar "x") (TmInt 1)

> tm

TmApp
  (TmLam "x" (Scope (TmAdd (TmVar (B ())) (TmInt 1))))
  (TmAdd (TmVar "x") (TmInt 1))

> tm >>= \v -> if v == "x" then TmInt 3 else TmVar v
```

```haskell
lam :: String -> Term String -> Term String
lam v tm = TmLam v (abstract1 v tm)
```

```
> let fn = lam "x" $ TmAdd (TmVar "x") (TmInt 1)

> fn

TmLam "x"
  (Scope (TmAdd (TmVar (B ())) (TmInt 1)))

> let tm = TmApp fn (TmAdd (TmVar "x") (TmInt 1)

> tm

TmApp
  (TmLam "x" (Scope (TmAdd (TmVar (B ())) (TmInt 1))))
  (TmAdd (TmVar "x") (TmInt 1))

> tm >>= \v -> if v == "x" then TmInt 3 else TmVar v

TmApp
  (TmLam "x" (Scope (TmAdd (TmVar (B ())) (TmInt 1))))
  (TmAdd (TmInt 3) (TmInt 1))
```

```haskell
eAppLam :: Rule (Term a) (Term a)
eAppLam _ (TmApp (TmLam _ s) tm) =
  pure $ instantiate1 tm s
eAppLam _ _ =
  Nothing
```

$$\frac{}{(\lambda\ x.t_{12})\ \ t_2 \longrightarrow [x \mapsto t_2]\ \ t_{12}}\ (\textsc{E-AppLam})$$

What can we do with lambda calculus?

We can do Booleans:

$$\begin{aligned} \mathit{tru} &= \lambda\ t.\ \lambda\ f.\ t \\ \mathit{fls} &= \lambda\ t.\ \lambda\ f.\ f \\ \mathit{and} &= \lambda\ b.\ \lambda\ c.\ b\ c\ \mathit{fls} \end{aligned}$$

We can do natural numbers:

$$\begin{aligned} z &= \lambda\ s.\ \lambda\ z.\ z \\ scc &= \lambda\ n.\ \lambda\ s.\ \lambda\ z.\ s\ \ (n\ s\ z) \\ plus\ m\ n &= \lambda\ s.\ \lambda\ z.\ m\ s\ \ (n\ s\ z) \end{aligned}$$

We can do pairs:

$$\begin{aligned} pair &= \lambda\ f.\ \lambda\ s.\ \lambda\ b.\ b\ f\ \ s \\ fst &= \lambda\ p.\ p\ tru \\ snd &= \lambda\ p.\ p\ fls \end{aligned}$$

$$fst\ \ (pair\ \ v\ \ w) \Rightarrow v$$

We even have enough to do recursion:

$$\mathit{fix} = \lambda\, f.\ (\lambda\, x.\ f\ \ (\lambda\, y.\ x\ x\ y))\ \ (\lambda\, x.\ f\ \ (\lambda\, y.\ x\ x\ y))$$

$$g = \lambda\ \mathit{fct}.\ \lambda\ n.\ \mathit{if}\ \ \mathit{eq}\ \ n\ 0\ \ \mathit{then}\ 1\ \ \mathit{else}\ \ \mathit{times}\ \ n\ \ (\mathit{fct}\ \ \mathit{prd}\ \ n)$$

$$\mathit{factorial} = \mathit{fix}\ g$$

Sometimes those kind of hijinx lead us into trouble:

$$omega = (\lambda\ x.\ x\ x)\,(\lambda\ x.\ x\ x)$$

$$omega \Rightarrow omega$$

One other big problem - there are plenty of stuck terms:

$$1\ 2$$

Simply Typed Lambda Calculus

Terms, values and types

t	$:=$	$\ldots$	
		x	*variable*
		$\lambda\ x : T\ .t$	*abstraction*
		$t\ \ t$	*function application*
v	$:=$	$\ldots$	
		$\lambda\ x : T\ .t$	*abstraction*
T	$:=$	$\ldots$	
		$T \to T$	*function arrow*

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

Terms, values and types

$$
\begin{array}{rcll}
t & := & \ldots & \\
 & & x & \textit{variable} \\
 & & \lambda\, x\, \boxed{:T}\,.t & \textit{abstraction} \\
 & & t\ \ t & \textit{function application} \\
v & := & \ldots & \\
 & & \lambda\, x\, \boxed{:T}\,.t & \textit{abstraction} \\
T & := & \ldots & \\
 & & \boxed{T \to T} & \textit{function arrow}
\end{array}
$$

We add *type annotations* to the variable bindings in our lambda terms.

Terms, values and types

$$
\begin{array}{lcll}
t & := & \dots & \\
 & & x & \textit{variable} \\
 & & \lambda\, x \colorbox{skyblue}{$:T$} .t & \textit{abstraction} \\
 & & t\ \ t & \textit{function application} \\
v & := & \dots & \\
 & & \lambda\, x \colorbox{skyblue}{$:T$} .t & \textit{abstraction} \\
T & := & \dots & \\
 & & \colorbox{skyblue}{$T \to T$} & \textit{function arrow}
\end{array}
$$

We also add an arrow type, that describes the type of functions.

Small-step semantics (eager)

$$\frac{t_1 \longrightarrow {t_1}'}{t_1\ t_2 \longrightarrow {t_1}'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow {t_2}'}{v_1\ t_2 \longrightarrow v_1\ {t_2}'} \quad \text{(E-App2)}$$

$$\frac{}{(\lambda\ x : T\ . t_1) t_2 \longrightarrow [x \mapsto t_2]\ t_1} \quad \text{(E-AppLam)}$$

Unsurprisingly, the small-step semantics don't change (except we now have type annotations).

Typing rules

$$\frac{}{\Gamma\,,\ x{:}T\ \vdash x{:}T} \qquad \text{(T-Var)}$$

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

$$\frac{\Gamma\,, x{:}T_1\ \vdash t{:}T_2}{\Gamma\ \vdash (\lambda\ x{:}T_1.t){:}T_1 \to T_2} \qquad \text{(T-Abs)}$$

Now we need a context for our typing rules.

Typing rules

$$\frac{}{\Gamma,\ x{:}T \vdash x{:}T} \qquad \text{(T-Var)}$$

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

$$\frac{\Gamma, x{:}T_1 \vdash t{:}T_2}{\Gamma \vdash (\lambda\ x{:}T_1.t){:}T_1 \to T_2} \qquad \text{(T-Abs)}$$

We use Γ as the context, which is a map from variables to types.

Typing rules

$$\frac{}{\Gamma\,,\, x{:}T \vdash x{:}T} \qquad (\text{T-Var})$$

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

$$\frac{\Gamma\,, x{:}T_1 \vdash t{:}T_2}{\Gamma \vdash (\lambda\ x{:}T_1.t) : T_1 \to T_2} \qquad (\text{T-Abs})$$

T-Var just grabs the type from the context.

Typing rules

$$\frac{}{\Gamma\,,\; x{:}T \;\vdash x{:}T} \qquad \text{(T-Var)}$$

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

$$\frac{\Gamma\,, x{:}T_1 \;\vdash t{:}T_2}{\Gamma \;\vdash (\lambda\; x{:}T_1.t){:}T_1 \to T_2} \qquad \text{(T-Abs)}$$

A type error occurs if the variable isn't found in the context.

Typing rules

$$\frac{}{\Gamma\,,\; x{:}T \;\vdash x{:}T} \qquad \text{(T-Var)}$$

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

$$\frac{\Gamma\,, x{:}T_1 \;\vdash t{:}T_2}{\Gamma \;\vdash (\lambda\; x{:}T_1.t){:}T_1 \to T_2} \qquad \text{(T-Abs)}$$

T-App has no new techniques in it.

Typing rules

$$\frac{}{\Gamma\,,\; x{:}T \;\vdash x{:}T} \qquad (\textsc{T-Var})$$

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

$$\frac{\Gamma\,, x{:}T_1 \vdash t{:}T_2}{\Gamma \;\vdash (\lambda\; x{:}T_1.t){:}T_1 \to T_2} \qquad (\textsc{T-Abs})$$

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

Typing rules

$$\frac{}{\Gamma\,,\ x{:}T\ \vdash x{:}T} \qquad \text{(T-VAR)}$$

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

$$\frac{\Gamma\,, x{:}T_1\ \vdash t{:}T_2}{\Gamma\ \vdash (\lambda\ x{:}T_1.t){:}T_1 \to\ T_2} \qquad \text{(T-ABS)}$$

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

Typing rules

$$\frac{}{\Gamma\,,\; x{:}T \;\vdash x{:}T} \qquad (\text{T-Var})$$

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

$$\frac{\Gamma\,, x{:}T_1 \;\vdash t{:}T_2}{\Gamma \;\vdash (\lambda\; x{:}T_1.t){:}T_1 \to T_2} \qquad (\text{T-Abs})$$

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

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \ \vdash\ x : \mathsf{Bool}}\ \text{T-Var}}{\emptyset \vdash \lambda\ x : \mathsf{Bool}\ .\ x\ :\ \mathsf{Bool} \to \mathsf{Bool}}\ \text{T-Abs} \qquad \dfrac{}{\emptyset \vdash \mathsf{true} : \mathsf{Bool}}\ \text{T-True}}{\emptyset \vdash (\lambda x{:}\mathsf{Bool}.x)\ \mathsf{true}\ : \mathsf{Bool}}\ \text{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \ \vdash \ x : \mathsf{Bool}}\ \text{T-Var}}{\emptyset \ \vdash \ \lambda \ x : \mathsf{Bool} \ . \ x \ : \ \mathsf{Bool} \ \to \ \mathsf{Bool}}\ \text{T-Abs} \qquad \dfrac{}{\emptyset \ \vdash \ \mathsf{true} : \mathsf{Bool}}\ \text{T-True}}{\emptyset \ \vdash \ (\ \lambda x{:}\mathsf{Bool}.x \ ) \ \mathsf{true} \ : \mathsf{Bool}}\ \text{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \ \vdash \ x : \mathsf{Bool}}\ \text{T-Var}}{\emptyset \ \vdash \ \lambda \ x : \mathsf{Bool} \ . \ x \ : \ \mathsf{Bool} \ \to \ \mathsf{Bool}}\ \text{T-Abs} \qquad \dfrac{}{\emptyset \ \vdash \ \mathsf{true} : \mathsf{Bool}}\ \text{T-True}}{\emptyset \ \vdash \ (\lambda x{:}\mathsf{Bool}.x)\ \mathsf{true} \ : \mathsf{Bool}}\ \text{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \ \vdash \ x : \mathsf{Bool}}\ \text{T-Var}}{\emptyset \vdash \lambda \ x : \mathsf{Bool} \ . \ x \ : \ \mathsf{Bool} \to \mathsf{Bool}}\ \text{T-Abs} \qquad \dfrac{}{\emptyset \vdash \mathsf{true} : \mathsf{Bool}}\ \text{T-True}}{\emptyset \vdash (\lambda x{:}\mathsf{Bool}.x)\ \mathsf{true} \ : \mathsf{Bool}}\ \text{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}}\ \text{T-Var}}{\emptyset \vdash \lambda\ x : \mathsf{Bool}\ .\ x : \mathsf{Bool} \to \mathsf{Bool}}\ \text{T-Abs} \qquad \dfrac{}{\emptyset \vdash \mathsf{true} : \mathsf{Bool}}\ \text{T-True}}{\emptyset \vdash (\lambda x{:}\mathsf{Bool}.x)\ \mathsf{true} : \mathsf{Bool}}\ \text{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \;\vdash\; x : \mathsf{Bool}}\;\text{T-Var}}{\emptyset \vdash \lambda\; x : \mathsf{Bool}\;.\; x \;:\; \mathsf{Bool} \to \mathsf{Bool}}\;\text{T-Abs} \qquad \dfrac{}{\emptyset \vdash \mathsf{true} : \mathsf{Bool}}\;\text{T-True}}{\emptyset \vdash (\lambda x{:}\mathsf{Bool}.x)\;\mathsf{true} \;:\; \mathsf{Bool}}\;\text{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \ \vdash \ x : \mathsf{Bool}}\ \text{T-Var}}{\emptyset \vdash \lambda\ x : \mathsf{Bool}\ .\ x : \mathsf{Bool} \to \mathsf{Bool}}\ \text{T-Abs} \qquad \dfrac{}{\emptyset \vdash \mathsf{true} : \mathsf{Bool}}\ \text{T-True}}{\emptyset \vdash (\lambda x{:}\mathsf{Bool}.x)\ \mathsf{true} : \mathsf{Bool}}\ \text{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}}\ \text{T-Var}}{\emptyset \vdash \lambda x : \mathsf{Bool} . x : \mathsf{Bool} \to \mathsf{Bool}}\ \text{T-Abs} \qquad \dfrac{}{\emptyset \vdash \mathsf{true} : \mathsf{Bool}}\ \text{T-True}}{\emptyset \vdash (\lambda x{:}\mathsf{Bool}.x)\ \mathsf{true} : \mathsf{Bool}}\ \text{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}}\ \textsf{T-Var}}{\emptyset \vdash \lambda x : \mathsf{Bool} . x : \mathsf{Bool} \to \mathsf{Bool}}\ \textsf{T-Abs} \qquad \dfrac{}{\emptyset \vdash \mathsf{true} : \mathsf{Bool}}\ \textsf{T-True}}{\emptyset \vdash (\lambda x{:}\mathsf{Bool}.x)\ \mathsf{true} : \mathsf{Bool}}\ \textsf{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}}\ \text{T-Var}}{\emptyset \vdash \lambda x : \mathsf{Bool} . x : \mathsf{Bool} \to \mathsf{Bool}}\ \text{T-Abs} \qquad \dfrac{}{\emptyset \vdash \mathsf{true} : \mathsf{Bool}}\ \text{T-True}}{\emptyset \vdash (\lambda x{:}\mathsf{Bool}.x)\ \mathsf{true} : \mathsf{Bool}}\ \text{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}}\ \text{T-Var}}{\emptyset \vdash \lambda x : \mathsf{Bool} . x : \mathsf{Bool} \to \mathsf{Bool}}\ \text{T-Abs} \qquad \dfrac{}{\emptyset \vdash \mathsf{true} : \mathsf{Bool}}\ \text{T-True}}{\emptyset \vdash (\lambda x{:}\mathsf{Bool}.x)\ \mathsf{true} : \mathsf{Bool}}\ \text{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}}\ \text{T-Var}}{\emptyset \vdash \lambda x : \mathsf{Bool} . x : \mathsf{Bool} \to \mathsf{Bool}}\ \text{T-Abs} \qquad \dfrac{}{\emptyset \vdash \mathsf{true} : \mathsf{Bool}}\ \text{T-True}}{\emptyset \vdash (\lambda x{:}\mathsf{Bool}.x)\ \mathsf{true} : \mathsf{Bool}}\ \text{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}}\ \text{T-Var}}{\emptyset \vdash \lambda\ x : \mathsf{Bool}\ .\ x : \mathsf{Bool} \to \mathsf{Bool}}\ \text{T-Abs} \qquad \dfrac{}{\emptyset \vdash \mathsf{true} : \mathsf{Bool}}\ \text{T-True}}{\emptyset \vdash (\lambda x{:}\mathsf{Bool}.x)\ \mathsf{true} : \mathsf{Bool}}\ \text{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}}\ \text{T-Var}}{\emptyset \vdash \lambda\ x : \mathsf{Bool}\ .\ x : \mathsf{Bool} \to \mathsf{Bool}}\ \text{T-Abs} \qquad \dfrac{}{\emptyset \vdash \mathsf{true} : \mathsf{Bool}}\ \text{T-True}}{\emptyset \vdash (\lambda x{:}\mathsf{Bool}.x)\ \mathsf{true} : \mathsf{Bool}}\ \text{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}}\;\text{T-Var}}{\emptyset \vdash \lambda\; x : \mathsf{Bool}\; .\; x : \mathsf{Bool} \to \mathsf{Bool}}\;\text{T-Abs} \qquad \dfrac{}{\emptyset \vdash \mathsf{true} : \mathsf{Bool}}\;\text{T-True}}{\emptyset \vdash (\lambda x{:}\mathsf{Bool}.x)\ \mathsf{true} : \mathsf{Bool}}\;\text{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}}\ \text{T-Var}}{\emptyset \vdash \lambda\ x : \mathsf{Bool}\ .\ x : \mathsf{Bool} \to \mathsf{Bool}}\ \text{T-Abs} \qquad \dfrac{}{\emptyset \vdash \mathsf{true} : \mathsf{Bool}}\ \text{T-True}}{\emptyset \vdash (\lambda x{:}\mathsf{Bool}.x)\ \mathsf{true} : \mathsf{Bool}}\ \text{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}}\ \text{T-Var}}{\emptyset \vdash \lambda\ x : \mathsf{Bool}\ .\ x : \mathsf{Bool} \to \mathsf{Bool}}\ \text{T-Abs} \qquad \dfrac{}{\emptyset \vdash \mathsf{true} : \mathsf{Bool}}\ \text{T-True}}{\emptyset \vdash (\lambda x{:}\mathsf{Bool}.x)\ \mathsf{true} : \mathsf{Bool}}\ \text{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}}\ \text{T-Var}}{\emptyset \vdash \lambda\ x : \mathsf{Bool}\ .\ x : \mathsf{Bool} \rightarrow \mathsf{Bool}}\ \text{T-Abs} \qquad \dfrac{}{\emptyset \vdash \mathsf{true} : \mathsf{Bool}}\ \text{T-True}}{\emptyset \vdash (\lambda x{:}\mathsf{Bool}.x)\ \mathsf{true} : \mathsf{Bool}}\ \text{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \text{Bool} \vdash x : \text{Bool}}\ \text{T-Var}}{\emptyset \vdash \lambda\ x : \text{Bool}\ .\ x : \text{Bool} \to \text{Bool}}\ \text{T-Abs} \qquad \dfrac{}{\emptyset \vdash \text{true} : \text{Bool}}\ \text{T-True}}{\emptyset \vdash (\ \lambda x{:}\text{Bool}.x\ )\ \text{true} : \text{Bool}}\ \text{T-App}$$

```haskell
inferVar :: Ord a => Rule (Context a, Term a) Type
inferVar _ (ctx, TmVar a) =
  fetchFromContext a ctx
inferVar _ _ =
  Nothing
```

$$\frac{}{\Gamma\,,\; x{:}T \;\vdash x{:}T} \qquad (\text{T-Var})$$

```haskell
inferApp :: Rule (Context a, Term a) Type
inferApp step (ctx, TmApp tm1 tm2) = do
  ty1 <- step (ctx, tm1)
  ty2 <- step (ctx, tm2)
  case ty1 of
    TyArr tyF tyT ->
      if ty2 == tyF
      then pure tyT
      else Nothing
    _ -> Nothing
inferApp _ _ =
  Nothing
```

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

```haskell
inferLam :: Rule (Context String, Term String) Type
inferLam step (ctx, TmLam v ty s) = do
  tyT <- step ( addToContext v ty ctx
              , instantiate1 (TmVar v) s
              )
  pure $ TyArr ty tyT
inferLam _ _ =
  Nothing
```

$$\frac{\Gamma\,, x{:}T_1 \;\vdash t{:}T_2}{\Gamma \;\vdash (\lambda\; x{:}T_1.t) : T_1 \to\; T_2}\;\text{(T-Abs)}$$

```haskell
checkVar :: Ord a => Rule (Context a, Term a, Type) ()
checkVar _ (ctx, TmVar v, ty) = do
  tyC <- fetchFromContext v ctx
  if tyC == ty then pure () else Nothing
checkVar _ _ =
  Nothing
```

$$\frac{}{\Gamma\,,\ x{:}T\ \vdash x{:}T}\qquad(\textsc{T-Var})$$

```haskell
checkApp :: RuleSet (Context String, Term String) Type
         -> Rule (Context String, Term String, Type) ()
checkApp infer step (ctx, TmApp tm1 tm2, ty) = do
  ty1 <- infer (ctx, tm1)
  case ty1 of
    TyArr tyF tyT -> do
      if tyT == ty
      then step (ctx, tm2, tyF)
      else Nothing
    _ ->
      Nothing
checkApp _ _ =
  Nothing
```

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

```haskell
checkLam :: Rule (Context String, Term String, Type) ()
checkLam step (ctx, TmLam v ty s, TyArr tyF tyT) = do
  if ty == tyF
  then do
    step ( addToContext v ty ctx
         , instantiate1 (TmVar v) s
         , tyT
         )
  else Nothing
checkLam _ _ =
  Nothing
```

$$\frac{\Gamma\,, x\!:\!T_1 \;\vdash t\!:\!T_2}{\Gamma \;\vdash (\lambda\; x\!:\!T_1.t) : T_1 \to\; T_2}\;(\text{T-ABS})$$

The typing relation rules out problematic terms like omega.

It also rules out fix - although we can add it back in later.

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

Finite terms will evaluate in a finite number of steps.

If we're happy to give that up, we can add fix back in.

Terms

$$\begin{array}{lcll} t & := & \ldots & \\ & & \textit{fix}\ t & \textit{fixed point} \end{array}$$

Small-step semantics

$$\frac{t \longrightarrow t'}{\mathit{fix}\ t \longrightarrow \mathit{fix}\ t'} \qquad \text{(E-Fix1)}$$

$$\frac{}{\mathit{fix}\,(\lambda\ x{:}T.t) \longrightarrow [x \mapsto \mathit{fix}\,(\lambda\ x{:}T.t)]\ t} \qquad \text{(E-FixBeta)}$$

Typing rules

$$\frac{\vdash t{:}T \to T}{\vdash \mathit{fix}\ t{:}T} \qquad (\textsc{T-Fix})$$

We can add all kinds of other things to STLC.

For example: pairs are straightforward to add.

Terms, values and types

$$\begin{array}{rcll}
t & := & \ldots & \\
 & & (t, t) & \textit{pair introduction} \\
 & & \textit{fst}\ t & \textit{pair elimination} \\
 & & \textit{snd}\ t & \textit{pair elimination} \\
v & := & \ldots & \\
 & & (v, v) & \textit{pair value} \\
T & := & \ldots & \\
 & & T \times T & \textit{pair type}
\end{array}$$

Small-step semantics

$$\frac{t_1 \longrightarrow t_1'}{(t_1, t_2) \longrightarrow (t_1', t_2)} \qquad \text{(E-Pair1)}$$

$$\frac{t_2 \longrightarrow t_2'}{(v_1, t_2) \longrightarrow (v_1, t_2')} \qquad \text{(E-Pair2)}$$

$$\frac{}{\mathit{fst}\,(v_1, v_2) \longrightarrow v_1} \qquad \text{(E-FstPair)}$$

$$\frac{}{\mathit{snd}\,(v_1, v_2) \longrightarrow v_2} \qquad \text{(E-SndPair)}$$

Typing rules

$$\frac{\vdash t_1 : T_1 \qquad \vdash t_2 : T_2}{\vdash (t_1, t_2) : T_1 \times T_2} \qquad \text{(T-Pair)}$$

$$\frac{\vdash (t_1, t_2) : T_1 \times T_2}{\vdash \mathit{fst}\ (t_1, t_2) : T_1} \qquad \text{(T-PairFst)}$$

$$\frac{\vdash (t_1, t_2) : T_1 \times T_2}{\vdash \mathit{snd}\ (t_1, t_2) : T_2} \qquad \text{(T-PairSnd)}$$

Tuples, records, variants and lists are similar.

There are still some things that are clunky.

We have to write a lot of different versions of id

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

$$\lambda\ x : \mathsf{Int}\ .\ x$$

Things are worse for const

$$\lambda\ x : \mathsf{Bool}\ .\ \lambda\ y : \mathsf{Bool}\ .\ x$$

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

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

$$\lambda\ x : \mathsf{Int}\ .\ \lambda\ y : \mathsf{Int}\ .\ x$$

Don't even get me started on compose

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

$$\ldots$$

We will address this soon...

Next time...

- Pattern matching

- Pattern matching
- Recursive types

- Pattern matching
- Recursive types
- User defined data types

- Pattern matching
- Recursive types
- User defined data types
- Hindley Milner

- Pattern matching
- Recursive types
- User defined data types
- Hindley Milner
- System F

- Pattern matching
- Recursive types
- User defined data types
- Hindley Milner
- System F
- System Fω

Code and slides are in `version-2` at

`https://github.com/dalaing/plt-talk`