Programming Language Theory

Dave Laing

BFPG March 2017

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

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

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

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

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

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

This talk is partly about getting you started with the notation, concepts and terminology.

It 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/or "Software Foundations" cover that very well if you are interested in that.

Natural deduction rules

We use natural deduction style rules for most of this.

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

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

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

We are dealing with an open world.

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

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

Integers

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

$$
\begin{array}{lll}
t & := & \\
 & \langle \mathsf{int} \rangle & \textit{constant integer} \\
 & t + t & \textit{addition}
\end{array}
$$

What do these terms look like?

3

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

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

What are the values for the Integer language?

By rules:

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

By definitions:

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

Evaluation proceeds in *steps*.

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

Small-step semantics

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

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

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

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

Small-step semantics

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

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

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

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

Small-step semantics

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

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

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

E-AddInt does the actual addition.

Small-step semantics

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

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

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

E-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

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

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

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

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

Small-step semantics

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

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

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

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

Small-step semantics

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

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

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

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

Small-step semantics

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

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

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

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

Small-step semantics

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

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

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

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

Small-step semantics

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

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

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

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

Small-step semantics

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

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

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

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

Small-step semantics

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

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

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

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

Small-step semantics

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

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

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

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

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

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

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

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

Previously...

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

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

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

Previously...

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

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

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

Previously...

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

$$\frac{}{\langle \mathsf{int}_1 \rangle \; + \; \langle \mathsf{int}_2 \rangle \longrightarrow \langle \mathsf{int}_1 + \mathsf{int}_2 \rangle} \quad \text{(E-ADDINT)}$$

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

Previously...

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

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

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

Previously...

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

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

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

Previously...

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

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

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

Previously...

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

$$\frac{}{\langle \mathsf{int}_1 \rangle \;+\; \langle \mathsf{int}_2 \rangle \longrightarrow \langle \mathsf{int}_1 + \mathsf{int}_2 \rangle} \qquad \text{(E-ADDINT)}$$

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

Previously...

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

$$\frac{}{\boxed{\langle \mathsf{int}_1 \rangle} + \boxed{\langle \mathsf{int}_2 \rangle} \longrightarrow \boxed{\langle \mathsf{int}_1 + \mathsf{int}_2 \rangle}} \quad \text{(E-ADDINT)}$$

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

The complete evaluation takes three steps.

First:

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

Then:

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

Finally:

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

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

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

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

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

$$\frac{}{\text{false or true} \longrightarrow \text{true}} \qquad \text{(E-ORFALSETRUE)}$$

$$\frac{}{\text{true or false} \longrightarrow \text{true}} \qquad \text{(E-ORTRUEFALSE)}$$

$$\frac{}{\text{true or true} \longrightarrow \text{true}} \qquad \text{(E-ORTRUETRUE)}$$

We should probably short-circuit the evaluation...

Small-step semantics

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

$$\frac{}{\text{false or } t_2 \longrightarrow t_2} \qquad \text{(E-ORFALSE)}$$

$$\frac{}{\text{true or } t_2 \longrightarrow \text{true}} \qquad \text{(E-ORTRUE)}$$

Small-step semantics

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

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

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

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

Small-step semantics

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

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

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

If the rules for the steps overlap for a particular term, but the result of the step is the same for all of the overlapping rules, then all is well.

Small-step semantics

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

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

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

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

Small-step semantics

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

$$\frac{}{\text{false or } t_2 \longrightarrow t_2} \qquad \text{(E-ORFALSE)}$$

$$\frac{}{\text{true or } t_2 \longrightarrow \text{true}} \qquad \text{(E-ORTRUE)}$$

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

Small-step semantics

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

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

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

If we had the short-circuiting rules and the non-short circuiting rules in use at the same time, we would have some troubles.

Small-step semantics

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

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

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

If we non-deterministically choose a rule to apply we will end up stepping to different terms, but evaluation will end up at the same value.

Small-step semantics

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

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

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

Things could be worse.

A non-deterministic set of rules

$$\frac{}{t = t} \quad (\textsc{Eq-Refl}) \qquad\qquad \frac{t_1 = t_2 \qquad t_3 = t_4}{t_1 \text{ or } t_3 = t_2 \text{ or } t_4} \quad (\textsc{Eq-Or})$$

$$\frac{t_2 = t_1}{t_1 = t_2} \quad (\textsc{Eq-Sym}) \qquad\qquad \frac{}{\text{false or } t = t} \quad (\textsc{Eq-OrFalse})$$

$$\frac{t_1 = t_2 \qquad t_2 = t_3}{t_1 = t_3} \quad (\textsc{Eq-Trans}) \qquad\qquad \frac{}{\text{true or } t = \text{true}} \quad (\textsc{Eq-OrTrue})$$

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

A non-deterministic set of rules

$$\frac{}{t = t} \quad \text{(Eq-Refl)}$$

$$\frac{t_2 = t_1}{t_1 = t_2} \quad \text{(Eq-Sym)}$$

$$\frac{t_1 = t_2 \qquad t_2 = t_3}{t_1 = t_3} \quad \text{(Eq-Trans)}$$

$$\frac{t_1 = t_2 \qquad t_3 = t_4}{t_1 \text{ or } t_3 = t_2 \text{ or } t_4} \quad \text{(Eq-Or)}$$

$$\frac{}{\text{false or } t = t} \quad \text{(Eq-OrFalse)}$$

$$\frac{}{\text{true or } t = \text{true}} \quad \text{(Eq-OrTrue)}$$

These are hard to implement, since you could get stuck applying

$$\frac{t_2 = t_1}{t_1 = t_2} \quad \text{(Eq-Sym)}$$

over and over.

A non-deterministic set of rules

$$\frac{}{t = t}\ (\textsc{Eq-Refl}) \qquad \frac{t_1 = t_2 \qquad t_3 = t_4}{t_1 \text{ or } t_3 = t_2 \text{ or } t_4}\ (\textsc{Eq-Or})$$

$$\frac{t_2 = t_1}{t_1 = t_2}\ (\textsc{Eq-Sym}) \qquad \frac{}{\mathsf{false} \text{ or } t = t}\ (\textsc{Eq-OrFalse})$$

$$\frac{t_1 = t_2 \qquad t_2 = t_3}{t_1 = t_3}\ (\textsc{Eq-Trans}) \qquad \frac{}{\mathsf{true} \text{ or } t = \mathsf{true}}\ (\textsc{Eq-OrTrue})$$

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

A non-deterministic set of rules

$$\frac{}{t = t} \quad (\textsc{Eq-Refl})$$

$$\frac{t_1 = t_2 \qquad t_3 = t_4}{t_1 \text{ or } t_3 = t_2 \text{ or } t_4} \quad (\textsc{Eq-Or})$$

$$\frac{t_2 = t_1}{t_1 = t_2} \quad (\textsc{Eq-Sym})$$

$$\frac{}{\text{false or } t = t} \quad (\textsc{Eq-OrFalse})$$

$$\frac{t_1 = t_2 \qquad t_2 = t_3}{t_1 = t_3} \quad (\textsc{Eq-Trans})$$

$$\frac{}{\text{true or } t = \text{true}} \quad (\textsc{Eq-OrTrue})$$

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

A non-deterministic set of rules

$$\frac{}{t = t} \quad (\textsc{Eq-Refl})$$

$$\frac{t_2 = t_1}{t_1 = t_2} \quad (\textsc{Eq-Sym})$$

$$\frac{t_1 = t_2 \qquad t_2 = t_3}{t_1 = t_3} \quad (\textsc{Eq-Trans})$$

$$\frac{t_1 = t_2 \qquad t_3 = t_4}{t_1 \text{ or } t_3 = t_2 \text{ or } t_4} \quad (\textsc{Eq-Or})$$

$$\frac{}{\text{false or } t = t} \quad (\textsc{Eq-OrFalse})$$

$$\frac{}{\text{true or } t = \text{true}} \quad (\textsc{Eq-OrTrue})$$

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

Natural numbers

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

This is more or less working in a unary number system

$$3 \equiv succ\ \ succ\ \ succ\ \ O$$

We can have eager natural numbers or lazy natural numbers

Terms and values (eager)

t	$:=$		
		O	*constant zero*
		$\mathsf{succ}\ t$	*successor*
		$\mathsf{pred}\ t$	*predecessor*
v	$:=$		
		O	*zero value*
		$\mathsf{succ}\ v$	*successor value*

Terms and values (lazy)

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

Small-step semantics

$$\frac{t_1 \longrightarrow {t_1}'}{\mathsf{succ}\ t_1 \longrightarrow \mathsf{succ}\ {t_1}'} \qquad (\text{E-S}\textsc{ucc}^*)$$

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

$$\frac{}{\mathsf{pred}\ \mathsf{O} \longrightarrow \mathsf{O}} \qquad (\text{E-P}\textsc{red}\text{Z}\textsc{ero})$$

$$\frac{}{\mathsf{pred}\ (\mathsf{succ}\ v) \longrightarrow v} \qquad (\text{E-P}\textsc{red}\text{S}\textsc{ucc}^{**})$$

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

These are the same natural numbers under eager evaluation:

$$\mathsf{succ}\ O$$

and under lazy evaluation:

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

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:

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

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.

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

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

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

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

* Uses t in place of v for the lazy version.

Small-step semantics for if

$$\frac{t_1 \longrightarrow t_1{}'}{\textsf{if } t_1 \textsf{ then } t_2 \textsf{ else } t_3 \longrightarrow \textsf{if } t_1{}' \textsf{ then } t_2 \textsf{ else } t_3} \qquad (\text{E-I}\textsc{f})$$

$$\frac{}{\textsf{if true then } t_2 \textsf{ else } t_3 \longrightarrow t_2} \qquad (\text{E-I}\textsc{f}\text{T}\textsc{rue})$$

$$\frac{}{\textsf{if false then } t_2 \textsf{ else } t_3 \longrightarrow t_3} \qquad (\text{E-I}\textsc{f}\text{F}\textsc{alse})$$

This term is stuck:

$$\mathsf{iszero\ false}$$

as is this one:

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

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

$$\frac{}{\mathsf{iszero}\ (\mathsf{succ}\ \boxed{nv}) \longrightarrow \mathsf{false}} \quad (\textsc{E-IsZeroSucc})$$

These terms are still stuck:

$$\text{iszero false}$$

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

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.

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

Enter *types*.

Types

$$
\begin{array}{lll}
\mathsf{T} & := & \\
 & \mathsf{Bool} & \text{type of booleans} \\
 & \mathsf{Nat} & \text{type of natural numbers}
\end{array}
$$

Typing rules (Booleans)

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

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

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

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

Typing rules (Booleans)

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

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

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

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

Typing rules (Booleans)

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

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

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

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

Typing rules (Booleans)

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

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

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

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

Typing rules (Booleans)

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

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

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

For now we don't need any more context.

Typing rules (Booleans)

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

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

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

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)

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

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

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

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)

$$\frac{}{\vdash \mathsf{O}{:}\mathsf{Nat}} \quad (\textsc{T-Zero})$$

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

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

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)

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

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

These rule out the stuck terms we saw previously:

$$\text{iszero false}$$

and

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

Typing rules (both)

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

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

They also rule out terms that are not stuck:

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

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

Typing rules (both)

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

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

This kind of thing is normally not a big deal.

Typing rules (both)

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

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

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.

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

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

For all well-typed terms,

the term is either a value or

is able to step .

Progress

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

For all well-typed terms,

the term is either a value or

is able to step .

Progress

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

For all well-typed terms,

the term is either a value or

is able to step .

Progress

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

For all well-typed terms,

the term is either a value or

is able to step .

Progress

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

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

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

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.

Lambda Calculus

This is where we step things up a notch

Terms and values

$$
\begin{array}{lcll}
t & := & \ldots & \\
 & & x & \textit{variable} \\
 & & \lambda\ x.t & \textit{abstraction} \\
 & & t\ \ t & \textit{function application} \\
v & := & \ldots & \\
 & & \lambda\ x.t & \textit{abstraction}
\end{array}
$$

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

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

Lambda anatomy

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

Lambda anatomy

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

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.

The term

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

is equivalent to

$$f(1)$$

when f is defined as before

In ordinary maths, we process

$$f(1)$$

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:

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

Small-step semantics

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

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

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

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.

$$\boxed{x} + 1$$

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.

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

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

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

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

Free variable rules

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

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

Free variable rules

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

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

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

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

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

Free variable rules

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

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

Free variable rules

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

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

Free variable rules

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

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

Free variable rules

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

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

Free variable rules

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

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

Free variable rules

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

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

Free variable rules

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

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

Free variable rules

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

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

Free variable rules

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

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

Free variable rules

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

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

Free variable rules

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

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

Free variable rules

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

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

Substitution rules

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

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

Substitution rules

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

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

Substitution rules

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

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

Substitution rules

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

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

Substitution rules

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

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

Substitution rules

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

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

Substitution rules

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

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

Substitution rules

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

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

Substitution rules

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

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

Substitution rules

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

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

Substitution rules

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

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

Substitution rules

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

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

Substitution rules

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

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

Substitution rules

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

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

Substitution rules

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

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

Substitution rules

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

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

Substitution rules

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

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

Substitution rules

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

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

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:

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

We can do pairs:

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

$$\mathit{fst}\ \ (\mathit{pair}\ \ v\ \ w) \Rightarrow v$$

We even have enough to do recursion:

$$fix = \lambda\ f.\ (\lambda\ x.\ f\ \ (\lambda\ y.\ x\ x\ y))\ \ (\lambda\ x.\ f\ \ (\lambda\ y.\ x\ x\ y))$$

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

$$factorial = fix\ g$$

Sometimes those kind of hijinx lead us into trouble:

$$\mathit{omega} = (\lambda\ x.\ x\ x)\,(\lambda\ x.\ x\ x)$$

$$\mathit{omega} \Rightarrow \mathit{omega}$$

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

$$1\ 2$$

Simply Typed Lambda Calculus

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 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

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

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

Small-step semantics

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

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

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

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

Typing rules

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

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

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

Now we need a context for our typing rules.

Typing rules

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

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

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

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

Typing rules

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

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

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

T-Var just grabs the type from the context.

Typing rules

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

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

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

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

Typing rules

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

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

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

T-App has no new techniques in it.

Typing rules

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

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

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

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

Typing rules

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

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

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

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

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

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

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

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

Typing rules

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

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

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

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

Typing rules

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

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

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

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

Typing rules

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

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

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

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

Typing rules

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

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

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

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

Typing rules

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

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

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

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

Typing rules

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

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

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

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

Typing rules

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

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

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

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

Typing rules

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

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

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

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

Typing rules

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

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

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

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

Typing rules

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

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

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

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

Typing rules

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

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

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

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

Typing rules

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

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

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

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

Typing rules

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

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

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

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

Typing rules

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

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

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

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

Typing rules

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

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

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

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

Typing rules

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

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

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

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

Typing rules

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

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

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

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

Typing rules

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

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

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

$$\cfrac{\cfrac{\cfrac{x : \text{Bool} \in x : \text{Bool}}{x{:}\text{Bool} \vdash x : \text{Bool}}\;\text{T-Var}}{\vdash \lambda \, x : \text{Bool} \, . \, x : \text{Bool} \rightarrow \text{Bool}}\;\text{T-Abs} \qquad \cfrac{}{\vdash \text{true} : \text{Bool}}\;\text{T-True}}{\vdash (\, \lambda x{:}\text{Bool}.x \,) \text{ true} : \text{Bool}}\;\text{T-App}$$

Typing rules

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

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

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

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

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

$$
\begin{array}{rcll}
t & := & \ldots & \\
 & & \mathit{fix}\ t & \mathit{fixed\ point}
\end{array}
$$

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

$$\frac{\vdash t{:}T \to T}{\vdash \mathit{fix}\ t{:}T} \quad (\text{T-Fix})$$

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

$$\frac{\vdash t_1 : T_1 \qquad \vdash t_2 : T_2}{\vdash (t_1, t_2) : T_1 \times T_2} \tag{T-Pair}$$

$$\frac{\vdash (t_1, t_2) : T_1 \times T_2}{\vdash \mathit{fst}\ (t_1, t_2) : T_1} \tag{T-PairFst}$$

$$\frac{\vdash (t_1, t_2) : T_1 \times T_2}{\vdash \mathit{snd}\ (t_1, t_2) : T_2} \tag{T-PairSnd}$$

Tuples, records, variants and lists are similar.

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

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.

Type inference for lambda calculus

We don't use types during evaluation.

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

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

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

There are some pathological cases with respect to running time, but it's normally fine.

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

The general idea is:

The general idea is:

- Generate type variables all over the place

The general idea is:

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

The general idea is:

- Generate type variables all over the place
- Record constraints on the type variables when we run into something concrete
- *Unify* these constraints to find a map from type variables to types

The general idea is:

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

Unification is a way of solving a symbolic computation.

Unification is a way of solving a symbolic computation.

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

Unification is a way of solving a symbolic computation.

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

Let us look at our old friend:

$$(\lambda\ x\ .\ x)\ \mathsf{true}$$

For

$$\lambda\, x\ .\ t$$

For

$$\lambda\ x\ .\ t$$

we expect that given

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

For

$$\lambda\, x\ .\ t$$

we expect that given

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

that the overall type will be

$$\lambda\, x\ .\ t : C \to\ D$$

For

$$\lambda\ x\ .\ t$$

we expect that given

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

that the overall type will be

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

with constraints

$$\emptyset$$

For

$$\lambda\ x\ .\ x$$

we expect that given

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

that the overall type will be

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

with constraints

$$\emptyset$$

For

$$\lambda\ x\ .\ x$$

we expect that given

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

that the overall type will be

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

with constraints

$$\emptyset$$

For

$$\lambda\ x\ .\ x$$

we expect that given

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

that the overall type will be

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

with constraints

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

For

$$\lambda\, x \;.\; x$$

we expect that given

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

that the overall type will be

$$\lambda\, x \;.\; x : C \to \; D$$

with constraints

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

For

$$\lambda\ x\ .\ x$$

we expect that given

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

that the overall type will be

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

with constraints

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

For

$$\lambda\ x\ .\ x$$

we expect that given

$$\Gamma, x : C \vdash x : \boxed{D}$$

that the overall type will be

$$\lambda\ x\ .\ x : C \to \boxed{D}$$

with constraints

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

For

$$\lambda\ x\ .\ x$$

we expect that given

$$\Gamma, x : C \vdash x : \boxed{C}$$

that the overall type will be

$$\lambda\ x\ .\ x : C \to \boxed{C}$$

with constraints

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

For

$$\lambda\ x\ .\ x$$

we expect that given

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

that the overall type will be

$$\lambda\ x\ .\ x : C \rightarrow\ C$$

with constraints

$$\emptyset$$

We know that

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

For

$$f \quad x$$

For

$$f\ x$$

we expect that given

$$\vdash f : A \to B$$

and

$$\vdash x : A$$

For

$$f\ x$$

we expect that given

$$\vdash f : A \to B$$

and

$$\vdash x : A$$

that the overall type will be

$$\vdash f\ x : B$$

For

$$f \ x$$

we expect that given

$$\vdash f : A \to B$$

and

$$\vdash x : A$$

that the overall type will be

$$\vdash f \ x : B$$

with constraints

$$\emptyset$$

For

$$(\lambda\ x\ .\ x)\ \ x$$

we expect that given

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

and

$$\vdash x :\ A$$

that the overall type will be

$$\vdash f\ \ x :\ B$$

with constraints

$$\emptyset$$

For

$$(\lambda\ x\ .\ x)\ \ x$$

we expect that given

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

and

$$\vdash x :\ A$$

that the overall type will be

$$\vdash f\ \ x :\ B$$

with constraints

$$\emptyset$$

For

$$(\lambda\ x\ .\ x)\ \ x$$

we expect that given

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

and

$$\vdash x :\ A$$

that the overall type will be

$$\vdash f\ \ x :\ B$$

with constraints

$$\left\{\left(\ A\ \rightarrow\ B\ \right) \sim \left(\ C\ \rightarrow\ C\ \right)\right\}$$

For

$$(\lambda\ x\ .\ x)\ \ x$$

we expect that given

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

and

$$\vdash x :\ A$$

that the overall type will be

$$\vdash f\ \ x :\ B$$

with constraints

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

For

$$(\lambda\ x\ .\ x)\ x$$

we expect that given

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

and

$$\vdash x\ :\ A$$

that the overall type will be

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

with constraints

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

For

$$(\lambda\ x\ .\ x)\ \text{true}$$

we expect that given

$$\vdash (\lambda\ x\ .\ x)\ :\ A\ \to\ B$$

and

$$\vdash x :\ A$$

that the overall type will be

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

with constraints

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

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

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

and

$$\vdash \mathsf{true}\ :\ \ A$$

that the overall type will be

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

with constraints

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

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

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

and

$$\vdash \mathsf{true} :\ A$$

that the overall type will be

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

with constraints

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

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

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

and

$$\vdash \mathsf{true} :\ A$$

that the overall type will be

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

with constraints

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

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

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

and

$$\vdash \mathsf{true} :\ A$$

that the overall type will be

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

with constraints

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

For

$$(\lambda\ x\ .\ x)\ \text{true}$$

we expect that given

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

and

$$\vdash \text{true} :\ A$$

that the overall type will be

$$\vdash (\lambda\ x\ .\ x)\ \text{true} :\ B$$

with constraints

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

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

$$\vdash (\lambda\ x\ .\ x)\ :\ A\ \to\ B$$

and

$$\vdash \mathsf{true} :\ A$$

that the overall type will be

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

with constraints

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

For

$$(\lambda\ x\ .\ x)\ \ \text{true}$$

we expect that given

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

and

$$\vdash \text{true} :\ A$$

that the overall type will be

$$\vdash (\lambda\ x\ .\ x)\ \ \text{true} :\ B$$

with constraints

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

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

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

and

$$\vdash \mathsf{true} :\ A$$

that the overall type will be

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

with constraints

$$\left\{\ A\ \sim\ B\ \sim\ C\ \sim \mathsf{Bool}\right\}$$

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

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

and

$$\vdash \mathsf{true} :\ A$$

that the overall type will be

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

with constraints

$$\left\{\ A\ \sim\ B\ \sim\ C\ \sim \mathsf{Bool}\right\}$$

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

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

and

$$\vdash \mathsf{true} :\ A$$

that the overall type will be

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

with constraints

$$\left\{ A \sim B \sim\ C\ \sim \mathsf{Bool} \right\}$$

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

$$\vdash (\lambda\ x\ .\ x)\ :\ \boxed{A} \to\ B$$

and

$$\vdash \mathsf{true} :\ A$$

that the overall type will be

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

with constraints

$$\left\{ \boxed{A} \sim \boxed{B} \sim\ C\ \sim \boxed{\mathsf{Bool}} \right\}$$

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

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

and

$$\vdash \mathsf{true} :\ A$$

that the overall type will be

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

with constraints

$$\left\{ A \sim B \sim\ C\ \sim \mathsf{Bool} \right\}$$

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

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

and

$$\vdash \mathsf{true}\ :\ A$$

that the overall type will be

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

with constraints

$$\left\{ A \sim B \sim\ C\ \sim \mathsf{Bool} \right\}$$

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

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

and

$$\vdash \mathsf{true} :\ A$$

that the overall type will be

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

with constraints

$$\left\{ A \sim B \sim\ C\ \sim \mathsf{Bool} \right\}$$

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

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

and

$$\vdash \mathsf{true} :\ \ A$$

that the overall type will be

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

with constraints

$$\left\{\boxed{A} \sim \boxed{B} \sim\ C\ \sim \boxed{\mathsf{Bool}}\right\}$$

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

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

and

$$\vdash \mathsf{true} :\ A$$

that the overall type will be

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

with constraints

$$\left\{ A \sim B \sim\ C\ \sim \mathsf{Bool} \right\}$$

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

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

and

$$\vdash \mathsf{true} :\ \boxed{A}$$

that the overall type will be

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

with constraints

$$\left\{ \boxed{A} \sim \boxed{B} \sim\ C\ \sim \boxed{\mathsf{Bool}} \right\}$$

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

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

and

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

that the overall type will be

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

with constraints

$$\left\{ A \sim B \sim\ C\ \sim \mathsf{Bool} \right\}$$

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

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

and

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

that the overall type will be

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

with constraints

$$\left\{ \boxed{A} \sim \boxed{B} \sim\ C\ \sim \boxed{\mathsf{Bool}} \right\}$$

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

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

and

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

that the overall type will be

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

with constraints

$$\left\{ A \sim B \sim\ C\ \sim \mathsf{Bool} \right\}$$

For

$$(\lambda\ x\ .\ x)\ \ \mathsf{true}$$

we expect that given

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

and

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

that the overall type will be

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

with constraints

$$\left\{ \boxed{A} \sim \boxed{B} \sim\ C\ \sim \boxed{\mathsf{Bool}} \right\}$$

For

$$(\lambda\ x\ .\ x)\ \ \text{true}$$

we expect that given

$$\vdash (\lambda\ x\ .\ x)\ :\ \text{Bool}\ \rightarrow\ \text{Bool}$$

and

$$\vdash \text{true} :\ \text{Bool}$$

that the overall type will be

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

with constraints

$$\emptyset$$

Let us revisit some terms that gave us trouble before.

$$\lambda\, x\, .\, x : \forall\, A\, .\, A \to A$$

What is going on with the $\forall A$?

That is a *universal type*.

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

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

You'll also see types with no variables in them being referred to as *monotypes* and types with variables in them being referred to as *polytypes*.

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

$$\lambda\, x\, .\, \lambda\, y\, .\, x : \forall\, A\, .\, \forall\, B\, .\, A \to B \to A$$

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

These things look really handy...

It would be nice to have access to them from within our language...

Universal types: System F

Types

$$
\begin{array}{rcll}
T & = & \dots & \\
 & & \boxed{X} & \textit{type variable} \\
 & & T \to T & \textit{function arrow} \\
 & & \boxed{\forall\, X.T} & \textit{univeral type}
\end{array}
$$

We start by adding universal types into the mix

Terms and values

$$
\begin{array}{lcll}
t & = & \ldots & \\
 & & x & \textit{variable} \\
 & & \lambda\ x{:}T.t & \textit{abstraction} \\
 & & t\ t & \textit{function application} \\
 & & \lambda\ X.t & \textit{type abstraction} \\
 & & t\ [T] & \textit{type application} \\
v & = & \ldots & \\
 & & \lambda\ x{:}T.t & \textit{abstraction} \\
 & & \lambda\ X.t & \textit{type abstraction}
\end{array}
$$

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

Terms and values

$$
\begin{array}{rcll}
t & = & \ldots & \\
 & & x & \textit{variable} \\
 & & \lambda\ x{:}T.t & \textit{abstraction} \\
 & & t\ \ t & \textit{function application} \\
 & & \lambda\ X.t & \textit{type abstraction} \\
 & & t\ \ [T] & \textit{type application} \\
v & = & \ldots & \\
 & & \lambda\ x{:}T.t & \textit{abstraction} \\
 & & \lambda\ X.t & \textit{type abstraction}
\end{array}
$$

We abstract over types to do things like id:

$$\lambda\ X\ .\ \lambda\ x : X\ .\ x : \forall\ A\ .\ A \to A$$

Terms and values

$$
\begin{array}{rcll}
t & = & \ldots & \\
 & & x & \textit{variable} \\
 & & \lambda\ x{:}T.t & \textit{abstraction} \\
 & & t\ \ t & \textit{function application} \\
 & & \lambda\ X.t & \textit{type abstraction} \\
 & & t\ \ [T] & \textit{type application} \\
v & = & \ldots & \\
 & & \lambda\ x{:}T.t & \textit{abstraction} \\
 & & \lambda\ X.t & \textit{type abstraction}
\end{array}
$$

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

$$(\lambda\ X\ .\ \lambda\ x : X\ .\ x)\ \ [\mathsf{Int}]\ : \mathsf{Int} \to \mathsf{Int}$$

Terms and values

$$
\begin{array}{lcll}
t & = & \ldots & \\
 & & x & \textit{variable} \\
 & & \lambda\ x{:}T.t & \textit{abstraction} \\
 & & t\ t & \textit{function application} \\
 & & \boxed{\lambda\ X.t} & \textit{type abstraction} \\
 & & \boxed{t\ [T]} & \textit{type application} \\
v & = & \ldots & \\
 & & \lambda\ x{:}T.t & \textit{abstraction} \\
 & & \boxed{\lambda\ X.t} & \textit{type abstraction}
\end{array}
$$

If we didn't have these things to guide the way, we'd lose syntax-directed type checking / inference.

Small-step semantics (old)

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

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

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

Small-step semantics (new)

$$\frac{t_1 \longrightarrow {t_1}'}{t_1\ \ [T_2] \longrightarrow {t_1}'\ \ [T_2]} \qquad \text{(E-TApp)}$$

$$\frac{}{(\lambda\ X.t_{12})\,[T_2] \longrightarrow [X \mapsto T_2]\, t_{12}} \qquad \text{(E-TAppTAbs)}$$

The new rules are straightforward.

Small-step semantics (new)

$$\frac{t_1 \longrightarrow t_1{}'}{t_1\ [T_2] \longrightarrow t_1{}'\ [T_2]} \quad \text{(E-TApp)}$$

$$\frac{}{(\lambda\, X.t_{12})\,[T_2] \longrightarrow [X \mapsto T_2]\,t_{12}} \quad \text{(E-TAppTAbs)}$$

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

Small-step semantics (new)

$$\frac{t_1 \longrightarrow t_1{}'}{t_1\ [T_2] \longrightarrow t_1{}'\ [T_2]} \qquad (\text{E-TApp})$$

$$\frac{}{(\lambda\, X.t_{12})\,[T_2] \longrightarrow [X \mapsto T_2]\, t_{12}} \qquad (\text{E-TAppTAbs})$$

E-TAppTAbs says that when we find a type application is applied to a type abstraction, we carry out the a substitution wherever that type appears in the term t_{i2}.

Small-step semantics (new)

$$\frac{t_1 \longrightarrow {t_1}'}{t_1\ \ [T_2] \longrightarrow {t_1}'\ \ [T_2]} \qquad \text{(E-TApp)}$$

$$\frac{}{(\lambda\ X.t_{12})\,[T_2] \longrightarrow [X \mapsto T_2]\,t_{12}} \qquad \text{(E-TAppTAbs)}$$

In this case that will be in the substitutions will be in the type annotations on the lambda abstractions.

Small-step semantics (new)

$$\frac{t_1 \longrightarrow {t_1}'}{t_1\ [T_2] \longrightarrow {t_1}'\ [T_2]} \qquad \text{(E-TApp)}$$

$$\frac{}{(\lambda\ X.t_{12})\,[T_2] \longrightarrow [X \mapsto T_2]\,t_{12}} \qquad \text{(E-TAppTAbs)}$$

For example:

$$(\lambda\ X\ .\ \lambda\ x : X\ .\ x)\ \ [\mathsf{Int}] \longrightarrow \lambda\ x : Int\ .\ x$$

Typing rules (old)

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

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

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

Typing rules (new)

$$\frac{\Gamma,\ X \vdash t_2 : T_2}{\Gamma \vdash (\lambda\ X\ .\ t_2\ ) : \forall\ X\ .\ T_2} \qquad (\text{T-TAbs})$$

$$\frac{\Gamma \vdash t_1 : \forall X\ .\ T_{12}}{\Gamma \vdash t_1[T_2] : [X \mapsto T_2]T_{12}} \qquad (\text{T-TApp})$$

T-TAbs gives type abstractions a universal type.

Typing rules (new)

$$\frac{\Gamma,\ X\ \vdash\ t_2 : T_2}{\Gamma \vdash (\lambda\ X\ .\ t_2\ ){:}\forall\ X\ .\ T_2} \qquad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash\ t_1 : \forall\ X\ .\ T_{12}}{\Gamma \vdash\ t_1[T_2] : [X \mapsto\ T_2]T_{12}} \qquad \text{(T-TApp)}$$

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

Typing rules (new)

$$\frac{\Gamma,\ X \vdash \boxed{t_2} : \boxed{T_2}}{\Gamma \vdash (\lambda\ X . \boxed{t_2}) : \forall\ X . \boxed{T_2}} \qquad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash\ t_1 : \forall X .\ T_{12}}{\Gamma \vdash\ t_1[T_2] : [X \mapsto T_2]T_{12}} \qquad \text{(T-TApp)}$$

Other than that, the type doesn't change.

Typing rules (new)

$$\frac{\Gamma, \boxed{X} \vdash t_2 : T_2}{\Gamma \vdash (\lambda X . t_2) : \forall X . T_2} \qquad (\text{T-TAbs})$$

$$\frac{\Gamma \vdash t_1 : \forall X . T_{12}}{\Gamma \vdash t_1[T_2] : [X \mapsto T_2]T_{12}} \qquad (\text{T-TApp})$$

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

Typing rules (new)

$$\frac{\Gamma,\ X\ \vdash\ t_2 : T_2}{\Gamma \vdash (\lambda\ \ X\ .\ t_2\ ):\forall\ \ X\ .\ T_2} \qquad (\text{T-TAbs})$$

$$\frac{\Gamma \vdash\ t_1 : \forall\ X\ .\ T_{12}}{\Gamma \vdash\ t_1[T_2] : [X \mapsto T_2]T_{12}} \qquad (\text{T-TApp})$$

T-TApp consumes the universal types.

Typing rules (new)

$$\frac{\Gamma,\ X\ \vdash\ t_2 : T_2}{\Gamma \vdash (\lambda\ X\ .\ t_2\ ):\forall\ X\ .\ T_2} \qquad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash\ t_1 : \forall\ X\ .\ T_{12}}{\Gamma \vdash\ t_1[T_2] : [X \mapsto T_2]T_{12}} \qquad \text{(T-TApp)}$$

We start with a term with a universal type.

Typing rules (new)

$$\frac{\Gamma,\ X \ \vdash\ t_2 : T_2}{\Gamma \vdash (\lambda\ \ X\ .\ t_2\ ){:}\forall\ \ X\ .\ T_2} \qquad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash\ t_1 : \forall\ X\ .\ T_{12}}{\Gamma \vdash\ t_1[T_2] : [X \mapsto\ T_2]T_{12}} \qquad \text{(T-TApp)}$$

We apply a type to that term.

Typing rules (new)

$$\frac{\Gamma,\ X \ \vdash\ t_2 : T_2}{\Gamma \vdash (\lambda\ \ X\ .\ t_2\ ){:}\forall\ \ X\ .\ T_2} \qquad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash\ t_1 : \forall\ X\ .\ T_{12}}{\Gamma \vdash\ t_1[T_2] : [X \mapsto T_2]T_{12}} \qquad \text{(T-TApp)}$$

And we get the type substitution that we were after.

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

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

The polymorphic version is more general, and so is the principal type.

We should check on how const and compose are doing.

$$\lambda\, X\ .\ \lambda\, Y\ .$$

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

$$x$$

$$: \forall\, A\ .\ \forall\, B\ .\ A \to B \to A$$

$$\lambda\, X\ .\ \lambda\, Y\ .\ \lambda\, Z\ .$$

$$\lambda\, f : Y \to Z\ .\ \lambda\, g : X \to Y\ .\ \lambda\, x : X\ .$$

$$f\ (g\ x)$$

$$: \forall\, A\ .\ \forall\, B\ .\ \forall\, C\ .\ (B \to C) \to (A \to B) \to (A \to C)$$

We can type them inside of our language, which is nice, but they are quite verbose.

To the type inference!

Type inference for System F is undecidable in general.

There is a whole new set of rabbit holes to dive into here.

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

There seems to be two main general approaches.

1. Use fancier type schemes - see ML^F, HMF, HML.

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

System F is still pretty nice.

It would be nice if we didn't have to roll our own pairs and things as type system extensions...

Universal types and type operators: System F_ω

We would like to be able to create a type Pair Int Bool without having to write support for pair types as part of our system.

This looks a little like a function taking two types and return a type.

These kind of things are referred to as *type operators*.

Types and kinds

$$
\begin{array}{rcll}
T & := & \ldots & \\
 & & X & \textit{type variable} \\
 & & T \to T & \textit{function arrow} \\
 & & \forall\, X \boxed{::K} .T & \textit{univeral type} \\
 & & \boxed{\lambda\, X::K.T} & \textit{operator abstraction} \\
 & & \boxed{T\ T} & \textit{operator application} \\
\boxed{K} & := & & \\
 & & * & \textit{kind of proper types} \\
 & & K \Rightarrow K & \textit{kind of type operators}
\end{array}
$$

We add operator abstraction to our set of types.

Types and kinds

$$
\begin{array}{rcll}
T & := & \ldots & \\
  &   & X & \textit{type variable} \\
  &   & T \to T & \textit{function arrow} \\
  &   & \forall\, X :: K . T & \textit{univeral type} \\
  &   & \lambda\, X :: K . T & \textit{operator abstraction} \\
  &   & T\ T & \textit{operator application} \\
K & := & & \\
  &   & * & \textit{kind of proper types} \\
  &   & K \Rightarrow K & \textit{kind of type operators}
\end{array}
$$

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

Types and kinds

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

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

Types and kinds

$$
\begin{array}{rcll}
T & := & \ldots & \\
 & & X & \textit{type variable} \\
 & & T \to T & \textit{function arrow} \\
 & & \forall\, X \,{::}K\,.T & \textit{univeral type} \\
 & & \lambda\, X{::}K.T & \textit{operator abstraction} \\
 & & T\ T & \textit{operator application} \\
K & := & & \\
 & & * & \textit{kind of proper types} \\
 & & K \Rightarrow K & \textit{kind of type operators}
\end{array}
$$

Pair Int Bool makes sense.

Types and kinds

T	$:=$	$\ldots$	
		X	*type variable*
		$T \to T$	*function arrow*
		$\forall\, X\, ::K\, .T$	*univeral type*
		$\lambda\, X::K.T$	*operator abstraction*
		$T\ T$	*operator application*
K	$:=$		
		$*$	*kind of proper types*
		$K \Rightarrow K$	*kind of type operators*

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

Types and kinds

$$
\begin{array}{rcll}
T & := & \ldots & \\
 & & X & \textit{type variable} \\
 & & T \to T & \textit{function arrow} \\
 & & \forall\, X\, \boxed{::K}\, .T & \textit{univeral type} \\
 & & \boxed{\lambda\, X::K.T} & \textit{operator abstraction} \\
 & & \boxed{T\ T} & \textit{operator application} \\
\boxed{K} & := & & \\
 & & * & \textit{kind of proper types} \\
 & & K \Rightarrow K & \textit{kind of type operators}
\end{array}
$$

Int Bool makes no sense at all.

Types and kinds

$$
\begin{array}{rcll}
T & := & \ldots & \\
 & & X & \textit{type variable} \\
 & & T \to T & \textit{function arrow} \\
 & & \forall\, X\, \boxed{::K}\,.T & \textit{univeral type} \\
 & & \boxed{\lambda\, X::K.T} & \textit{operator abstraction} \\
 & & \boxed{T\ T} & \textit{operator application} \\
\boxed{K} & := & & \\
 & & * & \textit{kind of proper types} \\
 & & K \Rightarrow K & \textit{kind of type operators}
\end{array}
$$

It should be ill-kinded and hence ruled out.

Terms and values

$$
\begin{array}{lcll}
t & := & \ldots & \\
 & & x & \textit{variable} \\
 & & \lambda\, x{:}T.t & \textit{abstraction} \\
 & & t\ t & \textit{function application} \\
 & & \lambda\, X\, \boxed{::K}\, .t & \textit{type abstraction} \\
 & & t\ [T] & \textit{type application} \\
v & := & \ldots & \\
 & & \lambda\, x{:}T.t & \textit{abstraction} \\
 & & \lambda\, X\, \boxed{::K}\, .t & \textit{type abstraction}
\end{array}
$$

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

Terms and values

$$
\begin{array}{rcll}
t & := & \ldots & \\
 & & x & \textit{variable} \\
 & & \lambda\, x{:}T.t & \textit{abstraction} \\
 & & t\ t & \textit{function application} \\
 & & \lambda\, X\, ::K\, .t & \textit{type abstraction} \\
 & & t\ [T] & \textit{type application} \\
v & := & \ldots & \\
 & & \lambda\, x{:}T.t & \textit{abstraction} \\
 & & \lambda\, X\, ::K\, .t & \textit{type abstraction}
\end{array}
$$

The same is true of the evaluation rules.

Typing rules (changes from STLC)

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

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

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

We need to make sure that our type annotations in lambda are types, rather than type operators.

Typing rules (changes from System F)

$$\frac{\Gamma, X \boxed{::K_1} \vdash t_2 : T_2}{\Gamma \vdash (\lambda X \boxed{::K_1}. t_2):\forall X \boxed{::K_1}. T_2} \qquad (\text{T-TAbs})$$

$$\frac{\Gamma \vdash t_1 : \forall X \boxed{::K_{11}}. T_{12} \qquad \boxed{\Gamma \vdash T_2::K_{11}}}{\Gamma \vdash t_1[T_2] : [X \mapsto T_2]T_{12}} \qquad (\text{T-TApp})$$

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

Typing rules (the new one)

$$\frac{\Gamma \vdash t{:}S \qquad S \equiv T \qquad \Gamma \vdash T{::}*}{\Gamma \vdash t{:}T} \qquad \text{(T-EQ)}$$

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

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

Typing rules (the new one)

$$\frac{\Gamma \vdash t{:}S \qquad S \equiv T \qquad \Gamma \vdash T{::}*}{\Gamma \vdash t{:}T} \qquad \text{(T-EQ)}$$

Given that, these all mean the same thing:

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

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

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

$$\mathsf{Id}\ (\mathsf{Nat} \to \mathsf{Bool})$$

Typing rules (the new one)

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

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

Typing rules (the new one)

$$\frac{\Gamma \vdash t{:}S \qquad S \equiv T \qquad \Gamma \vdash T{::}*}{\Gamma \vdash t{:}T} \qquad \text{(T-EQ)}$$

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

Typing equivalence rules (partial)

$$\frac{}{T \equiv T} \quad \text{(Q-REFL)}$$

$$\frac{T \equiv S}{S \equiv T} \quad \text{(Q-SYM)}$$

$$\frac{S \equiv T \qquad T \equiv U}{S \equiv U} \quad \text{(Q-TRANS)}$$

$$\frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{S_1 \to S_2 \equiv T_1 \to T_2} \quad \text{(Q-ARROW)}$$

...

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

Typing equivalence rules (partial)

$$\frac{}{T \equiv T} \qquad \text{(Q-REFL)}$$

$$\frac{T \equiv S}{S \equiv T} \qquad \text{(Q-SYM)}$$

$$\frac{S \equiv T \qquad T \equiv U}{S \equiv U} \qquad \text{(Q-TRANS)}$$

$$\frac{S_1 \equiv T_1 \qquad S_2 \equiv T_2}{S_1 \to S_2 \equiv T_1 \to T_2} \qquad \text{(Q-ARROW)}$$

...

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

Kinding rules

$$
\begin{array}{lcl}
T & := & \dots \\
 & & X \\
 & & \lambda\ X{::}K.T \\
 & & T\ \ T \\
K & := & \\
 & & K \Rightarrow K
\end{array}
$$

$$\frac{X{::}K \in \Gamma}{\Gamma \vdash X{::}K} \quad (\text{K-TVar})$$

$$\frac{\Gamma \vdash T_1{::}K_1 \to K_2 \qquad \Gamma \vdash T_2{::}K_1}{\Gamma \vdash T_1\ \ T_2{::}K_2} \quad (\text{K-App})$$

$$\frac{\Gamma, X{::}K_1 \vdash T_2{:}K_2}{\Gamma \vdash (\lambda\ X{::}K_1.T_2)\,{::}K_1 \to K_2} \quad (\text{K-Abs})$$

This looks familiar...

Kinding rules

$$
\begin{array}{rcl}
T & := & \dots \\
 & & X \\
 & & \lambda\ X{::}K.T \\
 & & T\ \ T \\
K & := & \\
 & & K \Rightarrow K
\end{array}
$$

$$\frac{X{::}K \in \Gamma}{\Gamma \vdash X{::}K} \quad \text{(K-TVar)}$$

$$\frac{\Gamma \vdash\ T_1{::}K_1 \rightarrow K_2 \qquad \Gamma \vdash\ T_2{::}K_1}{\Gamma \vdash\ T_1\ \ T_2{::}K_2} \text{(K-App)}$$

$$\frac{\Gamma, X{::}K_1 \vdash\ T_2{:}K_2}{\Gamma \vdash (\lambda\ X{::}K_1.T_2){::}K_1 \rightarrow K_2} \text{(K-Abs)}$$

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

Kinding rules

$$
\begin{array}{lcl}
T & := & \ldots \\
 & & X \\
 & & \lambda\, X{::}K.T \\
 & & T\ \ T \\
K & := & \\
 & & K \Rightarrow K
\end{array}
$$

$$\frac{X{::}K \in \Gamma}{\Gamma \vdash X{::}K} \quad \text{(K-TVar)}$$

$$\frac{\Gamma \vdash T_1{::}K_1 \to K_2 \qquad \Gamma \vdash T_2{::}K_1}{\Gamma \vdash T_1\ \ T_2{::}K_2} \quad \text{(K-App)}$$

$$\frac{\Gamma, X{::}K_1 \vdash T_2{:}K_2}{\Gamma \vdash (\lambda\, X{::}K_1.T_2){::}K_1 \to K_2} \quad \text{(K-Abs)}$$

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

Kinding rules

$$
\begin{array}{lll}
T & := & \dots \\
 & & X \\
 & & \lambda\, X{::}K.T \\
 & & T\ T \\
K & := & \\
 & & K \Rightarrow K
\end{array}
$$

$$\frac{X{::}K \in \Gamma}{\Gamma \vdash X{::}K} \quad (\textsc{K-TVar})$$

$$\frac{\Gamma \vdash T_1{::}K_1 \to K_2 \qquad \Gamma \vdash T_2{::}K_1}{\Gamma \vdash T_1\ T_2{::}K_2} \quad (\textsc{K-App})$$

$$\frac{\Gamma, X{::}K_1 \vdash T_2{:}K_2}{\Gamma \vdash (\lambda\, X{::}K_1.T_2){::}K_1 \to K_2} \quad (\textsc{K-Abs})$$

It also means we can drop the kind annotations and use our existing type inference algorithms to infer them!

Kinding rules

$$
\begin{array}{lcl}
T & := & \ldots \\
 & & X \\
 & & \lambda\ X{::}K.T \\
 & & T\ \ T \\
K & := & \\
 & & K \Rightarrow K
\end{array}
$$

$$\frac{X{::}K \in \Gamma}{\Gamma \vdash X{::}K} \quad (\text{K-TVar})$$

$$\frac{\Gamma \vdash T_1{::}K_1 \to K_2 \qquad \Gamma \vdash T_2{::}K_1}{\Gamma \vdash T_1\ \ T_2{::}K_2} \quad (\text{K-App})$$

$$\frac{\Gamma, X{::}K_1 \vdash T_2{:}K_2}{\Gamma \vdash (\lambda\ X{::}K_1.T_2){::}K_1 \to K_2} \quad (\text{K-Abs})$$

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

Kinding rules

$$
\begin{array}{rcl}
T & := & \dots \\
 & & X \\
 & & \lambda\ X{::}K.T \\
 & & T\ \ T \\
K & := & \\
 & & K \Rightarrow K
\end{array}
$$

$$\frac{X{::}K \in \Gamma}{\Gamma \vdash X{::}K} \quad (\text{K-TVar})$$

$$\frac{\Gamma \vdash T_1{::}K_1 \to K_2 \qquad \Gamma \vdash T_2{::}K_1}{\Gamma \vdash T_1\ \ T_2{::}K_2} \quad (\text{K-App})$$

$$\frac{\Gamma, X{::}K_1 \vdash T_2{:}K_2}{\Gamma \vdash (\lambda\ X{::}K_1.T_2){::}K_1 \to K_2} \quad (\text{K-Abs})$$

STLC is strongly normalizing...

Kinding rules

$$
\begin{array}{rcl}
T & := & \dots \\
 & & X \\
 & & \lambda\, X{::}K.T \\
 & & T\ \ T \\
K & := & \\
 & & K \Rightarrow K
\end{array}
$$

$$\frac{X{::}K \in \Gamma}{\Gamma \vdash X{::}K} \quad (\text{K-TVar})$$

$$\frac{\Gamma \vdash T_1{::}K_1 \to K_2 \qquad \Gamma \vdash T_2{::}K_1}{\Gamma \vdash T_1\ \ T_2{::}K_2} \ (\text{K-App})$$

$$\frac{\Gamma, X{::}K_1 \vdash T_2{:}K_2}{\Gamma \vdash (\lambda\ X{::}K_1.T_2)\,{::}K_1 \to K_2} \ (\text{K-Abs})$$

... so we can use STLC evaluation rules to normalize our types before we compare them, giving us algorithmic type equivalence.

Kinding rules

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \to T_2 :: *} \quad \text{(K-Arrow)}$$

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: *}{\Gamma \vdash \forall X :: K_1 . T_2 :: *} \quad \text{(K-All)}$$

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

Kinding rules

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \to T_2 :: *} \quad \text{(K-Arrow)}$$

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: *}{\Gamma \vdash \forall X :: K_1 . T_2 :: *} \quad \text{(K-All)}$$

Neither of these mess with the strong normalization property

Kinding rules

$$\frac{\Gamma \vdash T_1 :: * \qquad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \to T_2 :: *} \quad \text{(K-Arrow)}$$

$$\frac{\Gamma, X :: K_1 \vdash T_2 :: *}{\Gamma \vdash \forall X :: K_1 . T_2 :: *} \quad \text{(K-All)}$$

Hurrah!

What does Pair look like now?

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

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

Conclusion

Hopefully you understood some of that.

Sometimes it takes multiple passes before it clicks.

Plenty more in TAPL.

Possible future talks:

Possible future talks:

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

Possible future talks:

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

Possible future talks:

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

Possible future talks:

- Implementing interpreters and type systems for these (and other) languages.
- Funkier features: type classes, row polymorphism, linear types, and other fun.
- How to write compilers for these languages with LLVM.
- More about what you can do with unification.