COMPSCI 3MI3 - Principles of Programming Languages

Topic 8 - Simply Typed Lambda Calculus

NCC Moore

McMaster University

Fall 2021

Adapted from "Types and Programming Languages" by Benjamin C. Pierce

Introduction to Typed Lambda Calculus

TAE-k Some Constructive Criticism!

In the previous topic, we developed a type system for arithmetic expressions.

- Pros:
  - Well-typed programs were guaranteed to evaluate.
- Cons:
  - Untypeable programs were not guaranteed to not evaluate.
- Essentially, TAE was too conservative. It drew a boundary between "good" and "bad" terms, but in our efforts to keep all the bad terms out, we also kept out a lot of good terms as well.
- In this topic, we'll try to improve this result.

λ Gramma' !

In this topic, we will be considering a language containing the pure λ-Calculus covered in topics 5 and 6, enriched with Booleans. We will be using the following grammar.

$$
\begin{aligned}
\langle t \rangle ::= \;& \langle x \rangle \\
\mid \;& \lambda\ \langle x \rangle.\ \langle t \rangle \\
\mid \;& \langle t \rangle\ \langle t \rangle \\
\mid \;& \texttt{true} \\
\mid \;& \texttt{false} \\
\mid \;& \texttt{if}\ \langle t \rangle\ \texttt{then}\ \langle t \rangle\ \texttt{else}\ \langle t \rangle
\end{aligned}
$$

Where x is a variable in the λ-Calculus sense.

- We will intentionally exclude numbers to keep things simple for the time being.

Face Your Limitations!

Unfortunately, in Typed λ-Calculus (TLC), we will soon run into the same troubles we had in UAE. Consider the following (pseudo-) expression.

$$\texttt{if <some long computation> then true else } (\lambda x.x) \quad (1)$$

- The above expression remains untypeable, unless we evaluate that long, cumbersome expression.
- Remember, that in λ-Calculus, some expressions, such as the Ω-Function, **diverge!**
- Thus, we can't say for certain that typing the above expression is *even possible in a finite universe!*
  - A lot of languages (like Haskell) get around this problem by *requiring* the `then` and `else` cases to have the same type.

A (Preliminary) Set of Types

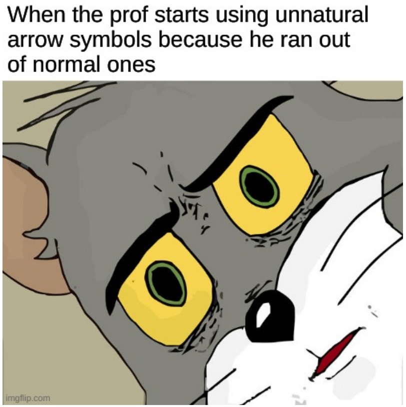
Our approach to typing in TLC will be very similar to TAE. We will define a typing relation mapping terms of our language to a set of types.

- So, we first need to define our set of types for TLC.

$$T = \{Bool, \Rightarrow\} \tag{2}$$

- The reason for including *Bool* should be obvious... but what the heck is with the arrow?
- It's obvious that we need some kind of type to represent functions, so we'll have $\Rightarrow$ stand in for now.
  - This will make more sense in a few slides.

It's Not Implication, That's Long Double Arrow


When the prof starts using unnatural arrow symbols because he ran out of normal ones
imgflip.com

- Implication
  - $\Longrightarrow$
- Evaluation Relation
  - $\rightarrow$
- Substitution
  - $\mapsto$
- **Function Type**
  - $\Rightarrow$

Green Arrow is Worse Batman

Let's give our new type some meaning, in the form of a typing rule!

$$\lambda x.t :\Rightarrow \tag{3}$$

Thus, every term with a λ has our new type. We can now determine the type of expressions such as:

$$\texttt{if true then } (\lambda x.\texttt{true}) \texttt{ else } (\lambda x.\lambda y.y) : \Rightarrow \tag{4}$$

Too Conservative!

Although this gives a sufficient description to use a TAE-style typing relation, $\Rightarrow$ currently tells us precious little about the functions themselves.

- $(\lambda x.\texttt{true})$ is a function which takes one argument and yields a Boolean.
- $(\lambda x.\lambda y.y)$ is a function which takes one argument and yields another function!

Things we generally like to know about functions:

- The number of arguments a function expects.
- The types of those arguments.
- The type of the value to be returned.

Refined Arrows

Let's refactor our definition of $\Rightarrow$ to include this information.

- Let's replace $\Rightarrow$ with an infinite family of types:

$$T_1 \Rightarrow T_2 \tag{5}$$

The above represents a function which expects an argument of type T_1, and yields a result of type T_2. Our set of types would then be generated by the following grammar.

$$\begin{aligned} \langle T \rangle ::= &\ \langle T \rangle \Rightarrow \langle T \rangle \\ \mid &\ \mathsf{Bool} \end{aligned}$$

Recursive Type Definitions!?

This grammar allows us to construct some really interesting types!

- $Bool \Rightarrow Bool$
  - A function mapping a Boolean argument to a Boolean result.
- $Bool \Rightarrow Bool \Rightarrow Bool$
  - A function mapping a Boolean argument to a function mapping a Boolean argument to a Boolean result.
  - $\Rightarrow$ is (funnily enough) **right associative**, so the above is equivalent to $Bool \Rightarrow (Bool \Rightarrow Bool)$
- $(Bool \Rightarrow Bool) \Rightarrow (Bool \Rightarrow Bool)$
  - A function mapping functions taking and returning a Boolean to functions taking and returning a Boolean.
- *Plus an infinite number of similar variations!*

Is This Looking Like Haskell Yet?

```haskell
func1 :: Bool -> Bool

func2 :: Bool -> Bool -> Bool

func3 :: (Bool -> Bool) -> (Bool -> Bool)
```

The Typing Relation

We know we want typing information for function inputs, but so far we have no idea how to get them. In general there are two approaches:

- **Explicit Typing** *(Used in this course)*.
  - Typing annotations are explicitly provided in the syntax of the function.
  - For TLC, we will annotate λ abstractions as follows:

$$\lambda x : T_1 . t_2 \tag{6}$$

- **Implicit Typing** *(Advanced topic in type theory)*.
  - We analyze the way a function is used to determine its typing information.

In reality, most languages use some amount of type inference, but for now, we will use explicit typing in TLC.

Function Typing Inference Rule

It should be obvious that the return type of a function of the form:

$$\lambda x : T_1 . t_2 \tag{7}$$

Is simply the type of the term t_2. We can thus construct an inference rule for typing λ abstractions.

$$\frac{t_2 : T_2}{(\lambda x : T_1 . t_2) : T_1 \Rightarrow T_2} \tag{8}$$

More Information Required!

There is, however, a problem with the foregoing inference mechanism. We do not yet have the ability to use typing information for abstracted variables to find the type of the inner term. Consider the following expression:

$$\lambda x : Bool.\ \texttt{if}\ x\ \texttt{then}\ s_2\ \texttt{else}\ s_3 \tag{9}$$

We'll call the inner term above t_2, and assume s_2 and s_3 have the same type T_2.

- If we consider only the information contained in t_2, this term is untypeable.
- x *must* be of type $Bool$ in order for t_2 to be well typed.
- We need some way to convey this information to t_2

Get Some Context!

On the previous slide, we consider $x : Bool$ to be the **context** in which we can say that the inner term (and therefore the entire function) is well-typed. We write this as:

$$x : Bool \vdash t_2 : T_2 \tag{10}$$

- In TLC, our typing relation becomes a **three-place relation**, taking elements from:
  - The set of terms.
  - The set of types.
  - The set of contexts.

$$\text{context} \vdash \text{term} : \text{type} \tag{11}$$

Don't Context This Point!

In general, we will often need to keep track of multiple context variables, because λ calculus has to manage arbitrarily nested abstraction operations.

- This means that the context we talked about in the previous slide is actually a **set** of variable typings, and is a subset of *all possible* variable typings. For example,

$$\{w : T_1, x : T_2, y : T_3\} \vdash z : T_4 \tag{12}$$

We generalize this form to:

$$\Gamma \vdash t : T \tag{13}$$

Where Γ is the set of variable type relations.

- This is often called either the **typing context** or the **typing environment**.

Gamma Rays!

More formally, Γ can be expressed as follows:

$$\Gamma = \begin{cases} \emptyset \\ \Gamma, x : T \end{cases} \tag{14}$$

That is to say, the typing context is either empty (∅), or composed by adding one piece of typing information to another context.

- We use a comma to add typing pairs to Γ.

If we are working with an empty context, we will often keep the turnstile character, but omit Γ. The following:

$$\vdash t_1 : T_1 \tag{15}$$

Means "In the absence of any other typing information, $t_1 : T_1$".

Function Typing, But Properly This Time

Using this information, we can refactor our typing rule for λ abstractions:

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash (\lambda x : T_1 . t_2) : T_1 \Rightarrow T_2} \qquad \text{(T-Abs)}$$

In humanspeak:

- Given a lambda abstraction $(\lambda x : T_1 . t_2)$, a typing context Γ,
- If we can conclude that $t_2 : T_2$ from both Γ and the typing information provided by x,
- We can also conclude that the type of the abstraction is $T_1 \Rightarrow T_2$.

Gamma Strain

Things of note:

- The way this is formulated, if begin with an empty Γ, and try to type nested *lambda* abstractions, we a new typing assumption to Γ for each λ deep we are in the derivation.
- Consider the following example.

$$\vdash \lambda x : Bool.\lambda y : Bool.\lambda z : Bool.y \tag{16}$$

Using Gammas

$$(1) \quad \boxed{\vdash \lambda x : Bool.\lambda y : Bool.\lambda z : Bool.y}$$

$$(2) \quad \boxed{\Gamma_1 \vdash \lambda y : Bool.\lambda z : Bool.y}$$

$$(3) \quad \boxed{\Gamma_2 \vdash \lambda z : Bool.y}$$

$$(4) \quad \boxed{\Gamma_3 \vdash y}$$

$$(5) \quad \Gamma_3 \vdash y : Bool$$

$$(6) \quad \Gamma_2 \vdash (\lambda z : Bool.y) : Bool \Rightarrow Bool$$

$$(7) \quad \Gamma_1 \vdash (\lambda y : Bool.\lambda z : Bool.y) : Bool \Rightarrow Bool \Rightarrow Bool$$

$$(8) \quad \vdash (\lambda x : Bool.\lambda y : Bool.\lambda z : Bool.y) : Bool \Rightarrow Bool \Rightarrow Bool \Rightarrow Bool$$

Gamma Contents

In the previous slide, the contents of the various typing contexts is as follows:

$$\Gamma_1 = \{x : \mathit{Bool}\} \tag{17}$$

$$\Gamma_2 = \{x : \mathit{Bool}, y : \mathit{Bool}\} \tag{18}$$

$$\Gamma_3 = \{x : \mathit{Bool}, y : \mathit{Bool}, z : \mathit{Bool}\} \tag{19}$$

Typing of Variables

You may have noticed in the previous derivation, we used a rule we haven't defined yet!

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

This rule simply states that, if a variable has a type association in Γ, we can pull that type out of Γ and type that variable with it.

- In order for this to work, however, we can't have duplicated variable names in Γ.
- We can use the same meta-rule that we used while describing substitution semantics. If a conflict occurs, we relabel the term with a non-conflicting variable name.

Submit Your Application!

To round off our terms in pure λ-Calculus, let's address function application.

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

Simply stated:

- If t_1 evaluates to a function mapping a term typed as T_1 to a term typed T_2,
- and the term provided as an argument has type T_1,
- we may conclude that the application of t_2 to t_1 has type T_2.

$\rightarrow$ *(typed)* *Based on λ (5-3)*

Syntax

$\mathtt{t}$	::=		*terms:*
		$\mathtt{x}$	*variable*
		$\lambda\mathtt{x}\mathtt{:T.t}$	*abstraction*
		$\mathtt{t}\ \mathtt{t}$	*application*
$\mathtt{v}$	::=		*values:*
		$\lambda\mathtt{x}\mathtt{:T.t}$	*abstraction value*
$\mathtt{T}$	::=		*types:*
		$\mathtt{T}\rightarrow\mathtt{T}$	*type of functions*
Γ	::=		*contexts:*
		$\varnothing$	*empty context*
		$\Gamma, \mathtt{x}\mathtt{:T}$	*term variable binding*

Evaluation $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \quad \text{(E-APP1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \quad \text{(E-APP2)}$$

$$(\lambda\mathtt{x}\mathtt{:T}_{11}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \quad \text{(E-APPABS)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

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

$$\frac{\Gamma, \mathtt{x}\mathtt{:T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda\mathtt{x}\mathtt{:T}_1.\mathtt{t}_2 : \mathtt{T}_1\rightarrow\mathtt{T}_2} \quad \text{(T-ABS)}$$

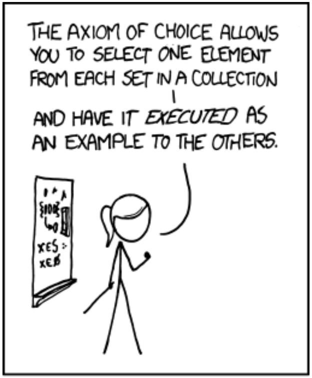
$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11}\rightarrow\mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \quad \text{(T-APP)}$$

Simply Typed **Pure** λ-Calculus

Things of note:

- In the foregoing slide, the highlighted parts are things that have been added, relative to our operational semantics of untyped pure λ calculus.
- Fun Fact: The pure type system is actually **degenerate**.
  - That is, there are *no valid well-typed terms* in this calculus.
- The reason is the way we are defining types themselves.
- We've recursively defined T, but no base case has been provided!
- This is why our examples so far have included the Booleans.

Properties of Typing


THE AXIOM OF CHOICE ALLOWS YOU TO SELECT ONE ELEMENT FROM EACH SET IN A COLLECTION
AND HAVE IT *EXECUTED* AS AN EXAMPLE TO THE OTHERS.

MY MATH TEACHER WAS A BIG BELIEVER IN PROOF BY INTIMIDATION.

Inversion Lemma

As with TAE, it will be useful to be able to derive the types of subterms of TLC from the typing information of the superterm. We will therefore introduce:

LEMMA [Inversion of the Typing Relation]

$$\Gamma \vdash x : R \implies x : R \in \Gamma \qquad \text{(I-Var)}$$

$$\begin{aligned} &\Gamma \vdash (\lambda x : T_1 . t_2) : R \\ \implies &\exists R_2 \mid R = T_1 \Rightarrow R_2 \land \Gamma, x : T_1 \vdash t_2 : R_2 \end{aligned} \qquad \text{(I-Abs)}$$

Inversion Aversion!

$$\Gamma \vdash t_1\ t_2 : R \implies \exists T_{11} \mid \Gamma \vdash t_1 : T_{11} \Rightarrow R \wedge \Gamma \vdash t_2 : T_{11} \quad \text{(I-App)}$$

$$\Gamma \vdash \mathit{true} : R \implies R = \mathit{Bool} \quad \text{(I-True)}$$

$$\Gamma \vdash \mathit{false} : R \implies R = \mathit{Bool} \quad \text{(I-False)}$$

$$\begin{aligned} &\Gamma \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : R \\ \implies &\Gamma \vdash t_1 : \mathit{Bool} \wedge \Gamma \vdash t_2 : R \wedge \Gamma \vdash t_3 : R \end{aligned} \quad \text{(I-If)}$$

Everyone is Unique!

Although we have chosen the path of explicit typing for TLC, so far we have only explicitly typed variables in function abstractions.

- It remains to be shown that this is sufficient to make the whole system typeable.
- One property that helps convince us of this is **uniqueness**.

THEOREM [Uniqueness of Types] In a given typing context Γ, if all the free variables of a term t are in the domain of Γ, t has at most one type.

Which Means No One Is Unique!

Another way of stating this is that if a term is typeable, its type is unique.

- Furthermore, there is only one derivation of this typing, which can be built from our typing relation's inference rules.

Proof Sketch:

- We essentially have to prove that not more than one typing rule can apply to any particular term at the same time.
- The observation that each typing rule applies to at maximum one term of the language is actually sufficient to demonstrate this property.
    - i.e., T-App only applies to terms of the form $t_1\,t_2$, etc.

It seems simple enough, but without this *nothing else works.*

Make Life Take the Lemmas Back!

Similarly to TAE, it will be useful to have canonical forms to refer to.

LEMMA [Canonical Forms]

1. If v is a value of type *Bool*, then v is either `true` or `false`.
2. If v is a value of type $T_1 \Rightarrow T_2$, then $v = \lambda x : T_1.t_2$.

Proof Sketch:

- This proof would proceed in the exact same manner as the canonical forms proof of TAE.
- That is, Demonstrate that the exclusivity of both categories with respect to the given values.

Proof of Progress

Proof of Progress I

The theorem of progress, as applied to TLC, is as follows:

THEOREM [Progress of Simply Typed λ-Calculus]

Suppose t is a well typed term with an empty typing context (that is, $\vdash t : T$ for some T). Either t is a value, or else there is some t' such that $t \rightarrow t'$.

The reason we are assuming an empty typing context here is that, if we can prove this for an empty typing context, we can add as much additional typing information as we like to Γ, and the property will still hold.

Proof by Induction on Typing Derivations :

- We will proceed by case analysis over all possible typing rules:
  - T-True; T-False; T-If; T-Var; T-Abs; T-App

Proof of Progress II

Let's address T-True, T-False and T-If at the same time.

- Recall our proofs of these properties from the progress theorem of TAE. Is there anything about TLC that would invalidate them?
- Nope! So let's save ourselves the time of repeating them.

What about T-Var?

- Consider the premise of T-Var: $x : T \in \Gamma$
- Remember that we assumed $\Gamma = \emptyset$
- Since nothing can be an element of $\emptyset$, T-Var can't be used under our current set of assumptions, so we don't have to worry about it!

Proof of Progress III

What about T-Abs?

- λ abstractions are values under TLC, so our theorem is trivially satisfied.

What about T-App? We can immediately state:

$$t = t_1\ t_2 \qquad \vdash t_1 : T_{11} \Rightarrow T_{12} \qquad \vdash t_2 : T_{11}$$

- Via the induction hypothesis, either $t_1 \in \mathcal{V}$ or $t_1 \rightarrow t_1'$.
- Also via the induction hypothesis, either $t_2 \in \mathcal{V}$ or $t_2 \rightarrow t_2'$.

Proof of Progress IV

	$t_1 \in \mathcal{V}$	$t_1 \to t_1'$
$t_2 \in \mathcal{V}$	E-AppAbs*	E-App1
$t_2 \to t_2'$	E-App2	E-App1

Applications of E-App1 and E-App2 are immediate from the evaluation rules, but E-AppAbs requires a bit more explanation.

- In order for E-AppAbs to be applicable, t_1 must be of the form $\lambda x : T_{11}.t_{12}$, and t_2 must be a value.
    - We know that $t_2 \in \mathcal{V}$
    - We can use clause 2 of the inversion lemma to deduce t_1 has the correct form, because we have previously stated that $t_1 : T_{11} \Rightarrow T_{12}$

Progress therefore holds for all typing rules of TLC. **QED**

Proof of Preservation

"Mr. Osborne, may I be excused? My brain is full."

Mutant Permutations!

In order to support the proof of preservation you will work together to create in tutorial next week, I'm going to throw some lemmas at you.

- The first states that the elements of a typing context Γ may be permuted without changing what is derivable from it.
- Recall that all the variables in Γ are presumed to be unique, and such uniqueness is guaranteed by relabelling as required.

LEMMA [PERMUTATION]
If $\Gamma \vdash t : T$ and Δ is a permutation of Γ, then $\Delta \vdash t : T$. Moreover, the latter derivation has the same depth as the former.

Proof Sketch: Straightforward induction on typing derivations.

Weak in the Knees

It can also be said that, if we can derive $t : T$ from some Γ, then we can add more items to Γ without changing this fact.

LEMMA [Weakening]
If $\Gamma \vdash t : T$ and $x \notin dom(\Gamma)$, then $\Gamma, x : S \vdash t : T$. Moreover, the latter derivation has the same depth as the former.

Proof Sketch: Straightforward induction on typing derivations.

- Both of the above lemmas should be fairly obvious and boring.
- They will help us prove that well-typedness is preserved when variables are substituted with terms of appropriate types.

This property is ubiquitous in the safety proofs of programming languages, where it is referred to as “the substitution lemma.”

Substitution Lemma I

LEMMA [Preservation of Types Under Substitution]

$$\Gamma, x : S \vdash t : T \land \Gamma \vdash s : S \implies \Gamma \vdash [x \mapsto s]t : T \qquad (20)$$

- Another way to state this is that, if we have some x which Γ types as S, and we have some s that has *the same type*, we can substitute x for s in t, without the type of t changing.
- Proof will proceed by induction over typing derivations, and using a case analysis over typing rules.

Just as a reminder:

$$\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda y.t_1) & = & \lambda y.\ [x \mapsto s]t_1 & \text{if } y \neq x \text{ and } y \notin FV(s) \\
[x \mapsto s](t_1\ t_2) & = & [x \mapsto s]t_1\ [x \mapsto s]t_2 &
\end{array}$$

Substitution Lemma II

Let's first consider T-True.

$$t = \texttt{true} \qquad\qquad T = Bool$$

- If t is either Boolean literal, it is unaffected by substitution operations.
- We can therefore conclude that $\Gamma \vdash [x \mapsto s]true : Bool$ is trivially true.

And of course, the same argument applies to T-False.

Substitution Lemma III

Consider T-If

$$t = \texttt{if}\ \ t_1\ \ \texttt{then}\ \ t_2\ \ \texttt{else}\ \ t_3$$

$$\Gamma, x : S \vdash t_1 : Bool \qquad \Gamma, x : S \vdash t_2 : T \qquad \Gamma, x : S \vdash t_3 : T$$

Using the induction hypothesis, we also have:

$$\Gamma \vdash [x \mapsto s]t_1 : Bool \qquad \Gamma \vdash [x \mapsto s]t_2 : T \qquad \Gamma \vdash [x \mapsto s]t_3 : T$$

From here, we can apply the typing rule T-If to conclude $t : T$. Thus, well-typedness is preserved under substitution in this case.

Substitution Lemma IV

Consider T-Var. We can immediately state:

$$t = z \qquad\qquad z : T \in (\Gamma, x : S)$$

- That is to say, we know t to be some variable (we've picked the name z for no particular reason).
- We know that z has type T, via the antecedent of the typing relation.

If we refer back to our substitution semantics, the only factor affecting the substitution of a variable for a term is whether that term is, in fact, the variable itself.

Substitution Lemma V

- Consider the case where $x = z$
  - $[x \mapsto s]z$ would then evaluate to s.
  - $x = z \wedge z = t \implies x = t$
  - Via the uniqueness of types, $x : S \wedge t : T \implies S = T$
  - Subbing these results into equation 17...

$$\Gamma, x : S \vdash x : S \wedge \Gamma \vdash s : S \implies \Gamma \vdash s : S \qquad (21)$$

  - Our conclusion is explicitly one of our premises!
- Now consider $x \neq z$
  - $[x \mapsto s]z$ would then evaluate to z (and from there to t).

$$\Gamma, x : S \vdash t : T \wedge \Gamma \vdash s : S \implies \Gamma \vdash t : T \qquad (22)$$

- By applying the weakening lemma to the conclusion, our conclusion has the first form as the first premise.

Substitution Lemma VI

Let's consider T-Abs. We immediately have:

$$t = \lambda y : T_2.t_1 \qquad T = T_2 \Rightarrow T_1 \qquad \Gamma, x : S, y : T_2 \vdash t_1 : T_1$$

By our meta-rule of substitutions in λ expressions, we derive:

$$x \neq y \qquad y \notin FV(s)$$

Using the the permutation lemma on the rightmost equation:

$$\Gamma, y : T_2, x : S \vdash t_1 : T_1 \tag{23}$$

Using the weakening lemma on $\Gamma \vdash s : S$:

$$\Gamma, y : T_2 \vdash s : S \tag{24}$$

By the induction hypothesis:

$$\Gamma, y : T_2 \vdash [x \mapsto s]t_1 : T_1. \tag{25}$$

Substitution Lemma VII

Recall T-Abs:

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \Rightarrow T_2} \tag{26}$$

Applying this to equation 22, we get:

$$\Gamma \vdash \lambda y : T_2 . [x \mapsto s] t_1 : T_2 \Rightarrow T_1 \tag{27}$$

The definition of substitution is:

$$[x \mapsto s](\lambda y : T_2 . t_1) = \lambda y : T_2 . [x \mapsto s] t_1 \tag{28}$$

- The left-hand side is of type $T_2 \Rightarrow T_1$ from our original case analysis.
- The right-hand side has the same type via equations 20 - 24.

Therefore, well-typedness is preserved by substitution in this case.

Substitution Lemma VIII

Finally, let's consider T-App. We immediately have:

$$t = t_1\ t_2 \qquad \Gamma, x : S \vdash t_1 : T_2 \Rightarrow T_1 \qquad \Gamma, x : S \vdash t_2 : T_2$$

$$T = T_1$$

Via the induction hypothesis, we also have:

$$\Gamma \vdash [x \mapsto s]t_1 : T_2 \Rightarrow T_1 \qquad \Gamma \vdash [x \mapsto s]t_2 : T_2$$

Substitution Lemma IX

Recall the typing rule for application (with some slight renaming):

$$\frac{\Gamma \vdash t_1 : T_2 \Rightarrow T \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T} \tag{29}$$

And recall our substitution rule for application...

$$[x \mapsto s](t_1\ t_2) = [x \mapsto s]t_1\ [x \mapsto s]t_2 \tag{30}$$

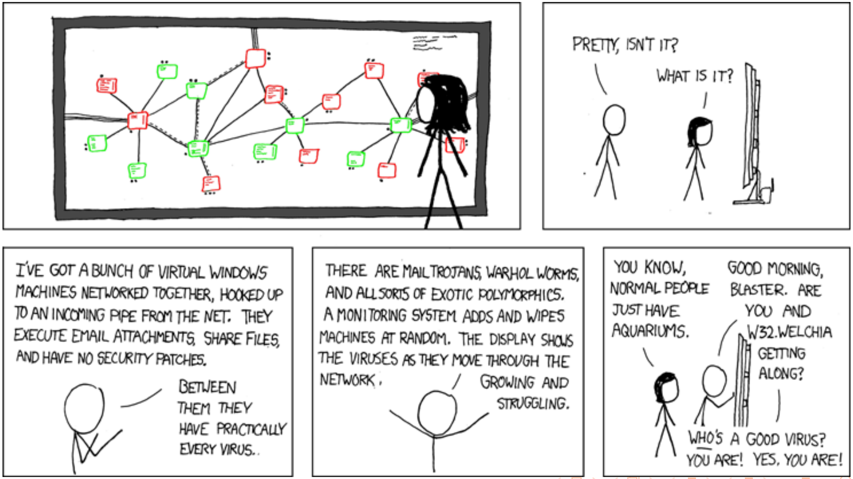
By applying T-App to our induction hypotheses, we determine that:

$$\Gamma \vdash ([x \mapsto s]t_1\ [x \mapsto s]t_2) : T \tag{31}$$

Thus, typing is preserved over substitution in the case of function application.

- **QED!**

Last Slide Comic


PRETTY, ISN'T IT?
WHAT IS IT?
I'VE GOT A BUNCH OF VIRTUAL WINDOWS MACHINES NETWORKED TOGETHER, HOOKED UP TO AN INCOMING PIPE FROM THE NET. THEY EXECUTE EMAIL ATTACHMENTS, SHARE FILES, AND HAVE NO SECURITY PATCHES.
BETWEEN THEM THEY HAVE PRACTICALLY EVERY VIRUS.
THERE ARE MAIL TROJANS, WARHOL WORMS, AND ALL SORTS OF EXOTIC POLYMORPHICS. A MONITORING SYSTEM ADDS AND WIPES MACHINES AT RANDOM. THE DISPLAY SHOWS THE VIRUSES AS THEY MOVE THROUGH THE NETWORK,
GROWING AND STRUGGLING.
YOU KNOW, NORMAL PEOPLE JUST HAVE AQUARIUMS.
GOOD MORNING, BLASTER. ARE YOU AND W32.WELCHIA GETTING ALONG?
WHO'S A GOOD VIRUS? YOU ARE! YES, YOU ARE!