

COMP 321 Lecture 09/Lecture 10: Type Systems, Progress and Preservation

Dan Licata

Fall 2026

1 Type Systems

In the language IMP that we have discussed up until this point, when does your implementation of stepping in SML raise a `StepError`?

- a function call $f(e)$ with a function f that is not in the program
- a function call $f(e_1, \dots, e_n)$ with a different number of arguments than the function declaration $f(\text{var } x_1, \dots, \text{var } x_m)$
- a use of an undeclared variable x
- a use of `get` on an address that is not in the domain of the memory μ

In all of these cases, the operational semantics we have given is *undefined*, in the sense that these expressions/statements are not done and also do not take a step by the stepping rules $\mapsto$. For example, consider the rule

$$\frac{a \in \text{dom}(\mu)}{\mu \mid \text{Get}(\text{Num}(a)) \mapsto \text{Num}(\text{lookup}(\mu, a))}$$

which only applies when a is in the domain of the memory μ (the address that are supposed to be allocated and available to the program). Then $\text{Get}(\text{Num}(b))$ for b not in the domain of the memory μ has undefined behavior. This means the *implementation* of the language has to choose a behavior that is not *specified* by the language definition. For example, the implementation might do something C-like, and choose to return whatever is in the underlying array of the memory at that address if there happens to be something there.

Here are some options for dealing with this situation:

- Allow the language implementation to choose what to do. Pros: flexible, e.g. the implementation can implement the undefined programs in a way that speeds up the defined programs, like how in C memory accesses are not bounds-checked. Cons: not portable — the program might work on one compiler but not another. Also often not secure — the undefined behavior can lead to security bugs, like accessing memory the program is not supposed to.

As an alternative to leaving the behavior completely unspecified by the operational semantics, we could make the behavior formally defined to be “undefined”. For example, we could add a rule

$$\frac{a \notin \text{dom}(\mu)}{\mu \mid \text{Get}(\text{Num}(a)) \mapsto \text{Num}(v)}$$

which says that the program is allowed to step to *any* value v it wants to, and so in different traces could step to different values. This has mostly the same advantages and disadvantages, but at least is clearer about what the programming language's intention is.

- Silently step to some default value. For example,

$$\frac{a \notin \text{dom}(\mu)}{\mu \mid \text{Get}(\text{Num}(a)) \mapsto \text{Num}(0)}$$

For function calls with the wrong number of arguments, we might do

$$\frac{f(\text{var } x_1, \dots, \text{var } x_m)\{s\} \in p \quad m \neq n}{\mu \mid f(e_1, \dots, e_n) \mapsto \mu \mid \text{Noop}}$$

Pros: now portable. Cons: Can't tell a successful program that actually finds a 0 in memory from an error; errors will “snowball” because the default values will propagate later in the program until they eventually cause a crash or incorrect value.

- Make the program raise an exception inside the language (not in the meta-language). For example,

$$\frac{a \notin \text{dom}(\mu)}{\mu \mid \text{Get}(\text{Num}(a)) \mapsto \text{raiseDomain}} \quad \frac{f(\text{var } x_1, \dots, \text{var } x_m)\{s\} \in p \quad m \neq n}{\mu \mid f(e_1, \dots, e_n) \mapsto \mu \mid \text{raiseArgs}}$$

This is what many dynamically typed languages like Lisp, Scheme, and Python do. Pros: the errors are detectable by a program, so the program can recover from the error. Cons: the errors are not caught until you run the program.

- Try to catch the errors when the programmer writes the program (*at compile-time / statically*) before the program is run by the user (*at run-time / dynamically*). Pros: finds problems faster. Cons: rules out some actually fine programs too.

2 Type Systems

To catch the undefined programs mentioned above statically, we will use a *type system*. The basic idea of the type system is a relation $e : \tau$, which means that e has type τ , and another relation $s \text{ ok}$, which means that the statement s is well-formed. Then the idea is that we will only step an expression e using the operational semantics after first having checked that it has some type $e : \tau$. However, to catch uses of undefined variables, we will also need to keep track of which variables are in scope. We do this using a *context* written as Γ (the Greek letter capital gamma), where Γ is a list of variable/type pairs written $x_1 : \tau_1, \dots, x_n : \tau_n$. This means that the variables x_i are in scope and have types τ_i . Then the two typing relations become

- $\Gamma \vdash e : \tau$ (in context Γ , the expression e has type τ)
- $\Gamma \vdash s \text{ ok}$ (in context Γ , the statement s is well-formed)

For IMP, we will have types $\text{int}, \text{int}^*, (\text{int}^*)^*$ etc, which we defined by the following grammar:

$$\tau ::= \text{int} \mid \tau^*$$

2.1 Typing Rules for Expressions

The type systems are specified by inference rules, with one inference rule saying when each syntactic form of expression/statement is well-typed. Here are the typing rules for expressions:

$$\begin{array}{c}
 \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \\
 \\
 \frac{}{\Gamma \vdash \text{Num}(a) : \text{int}} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash \text{Plus}(e_1, e_2) : \text{int}} \\
 \\
 \frac{\Gamma \vdash e : \tau^*}{\Gamma \vdash \text{Get}(e) : \tau}
 \end{array}$$

The first rule says that a variable x has whatever type it is declared to have in the context.

The second rule says that a numeric constant has type `int`. For now, this type system will exclude programs that use specific numbers as memory addresses (e.g. `get(12)`), and programs that do memory address arithmetic (e.g. `get(x + 7)` for getting the 7th element of an array) and ensure that all `gets` are on allocated addresses that are in the domain of the memory. (To include these programs, we could change the type system to not distinguish the types `int` and `int*`, but this would be less clear as an introduction to type systems.)

The third rule says that the `Plus` operation takes two expression of type `int` as input and outputs an `int`.

The fourth rule says that the `Get` operation takes something of pointer type as input and outputs whatever type that pointer contains.

2.2 Typing Derivations

The way we prove that an expression is well-typed is by giving a *typing derivation*, which is a tree built using the inference rules that shows that each sub-expression of the program is well-typed. For example:

$$\frac{\frac{x : \text{int}^*, y : \text{int}^* \vdash x : \text{int}^*}{x : \text{int}^*, y : \text{int}^* \vdash \text{Get}(x) : \text{int}} \quad \frac{x : \text{int}^*, y : \text{int}^* \vdash y : \text{int}^*}{x : \text{int}^*, y : \text{int}^* \vdash \text{Get}(y) : \text{int}}}{x : \text{int}^*, y : \text{int}^* \vdash \text{Plus}(\text{Get}(x), \text{Get}(y)) : \text{int}}$$

When there is no typing derivation $\Gamma \vdash e : \tau$, we say that e is ill-typed. For example, the following programs are ill-typed:

- Variable not in scope $x : \text{int}^*, y : \text{int}^* \not\vdash z : \tau$
- Too few dereferences $x : \text{int}^*, y : \text{int}^* \not\vdash \text{Plus}(\text{Get}(x), y) : \tau$
- Too many dereferences $x : \text{int}^* \not\vdash \text{Get}(\text{Get}(x)) : \tau$

2.3 Typing Rules for Statements

Next, we will say how to type check statements:

$$\begin{array}{c}
 \frac{\Gamma \vdash e_1 : \tau^* \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{Set}(e_1, e_2) \text{ ok}} \\
 \\
 \frac{\Gamma \vdash s_1 \text{ ok} \quad \Gamma \vdash s_2 \text{ ok}}{\Gamma \vdash \text{Sequence}(s_1, s_2) \text{ ok}} \\
 \\
 \frac{f(\tau_1 x_1, \dots, \tau_n x_n)\{s\} \in p \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \text{Call}(f, e_1, \dots, e_n) \text{ ok}} \\
 \\
 \frac{\Gamma, x : \tau^* \vdash s \text{ ok}}{\Gamma \vdash \text{LocalVar}(x : \tau.s) \text{ ok}}
 \end{array}$$

The first rule says that an assignment $e_1 := e_2$ is well-typed when e_1 has some pointer type and e_2 has the type that should be contained in the memory that such a pointer addresses.

The second rule says that a sequence of statements is well-typed if both statements are individually well-typed.

For the next two rules, we have changed the syntax to declare the intended types of variables in function arguments and in local variable declarations. For example, instead of writing $\{\text{var } x; s\}$ we would write $\{\text{int } x; s\}$ or $\{\text{int}^* x; s\}$.

The third rule says that a function call is well-typed if the actual number of arguments is the same as the number of variables in the function declaration, and if the arguments have the types specified in the function declaration.

The fourth rule says that a local variable declaration is well-typed if the body of the declaration is well-typed with the newly declared variable added to the context so that it is in scope.

For the types of variables, our convention is that we write the type of the *contents* of the memory addressed by that variable, so the variable itself has an extra level of pointer type. For example, consider the program

```

{int x;
 x := 1;
 x := get(x) + 1;
}

```

The local variable x will be substituted with a newly allocated memory address, and is used as the left-hand side of an assignment and as the input to a `get`, so must have a pointer type int^* . This is because all variables in IMP are *mutable*. A different convention would be to write $\text{int}^* x$ and insist that the types of all variable declarations are annotated with a pointer type τ^* , but the convention we use looks more like C syntax.

We also need to check that the program $p ::= f_1 \dots f_m$ is a list of well-typed function declarations, where we type check each function declaration by the following rule:

$$\frac{x_1 : \tau_1^*, \dots, x_n : \tau_n^* \vdash s \text{ ok}}{f(\tau_1 x_1, \dots, \tau_n x_n)\{s\} \text{ ok}}$$

This says that the body of the function must be well-typed referring only to the function argument variables, where each local variable again gets a pointer type.

2.4 Progress

We would like to prove that this type system rules out the undefined programs that we identified above. Intuitively, the type system checks that all variables are in the context when they are used, the type system checks that all functions are declared in the program, and the type system checks that a function has the correct number of arguments for the function's declaration, so the type system should rule out programs that don't obey those constraints. The type system also checks that `get` is only performed on something of declared pointer type (not an explicit address or one calculated by pointer arithmetic), so the only way to create an address is via a local variable, which allocates memory, and there is no explicit freeing of memory. This means `get` should only be called on addresses that are in memory.

We will work up to proving formally that there are no programs with undefined behavior allowed by the type system, but to introduce the techniques we will start with a simpler language without memory.

3 Immutable Variables

Up until now, we have studied an imperative language IMP that is in the style of C. To introduce proofs about type systems, it is simpler to switch to a purely functional language. Eventually we will re-integrate this with imperative features.

In a purely functional language, there is no need for statements or an explicit memory μ , so the operational semantics consists of just a single relation for stepping expressions $e \mapsto e'$. The functional language we consider today will have two types, `int` and `bool`, and write τ for an arbitrary type. We have the following expressions:

$$e ::= x \mid \text{Num}(k) \mid \text{Plus}(e_1, e_2) \mid \text{True} \mid \text{False} \mid \text{If}(e, e_1, e_2) \mid \text{Let}(e_1, x.e_2)$$

In concrete syntax, we write

```
e ::= x | k | e1 + e2
    | true | false | if e then e1 else e2
    | let x = e1 in e2
```

In class, we often/usually write the operational semantics and typing rules using the concrete syntax notation, but officially they are defined on abstract syntax trees.

The operational semantics should mostly be familiar:

$$\begin{array}{c}
 \overline{\text{Num}(k) \text{ done}} \\
 \\
 \frac{e_1 \mapsto e'_1}{\text{Plus}(e_1, e_2) \mapsto \text{Plus}(e'_1, e_2)} \quad \frac{e_2 \mapsto e'_2 \text{ e1 done}}{\text{Plus}(e_1, e_2) \mapsto \text{Plus}(e_1, e'_2)} \quad \frac{}{\text{Plus}(\text{Num}(k_1), \text{Num}(k_2)) \mapsto \text{Num}(k_1 + k_2)} \\
 \\
 \overline{\text{True done}} \quad \overline{\text{False done}} \\
 \\
 \frac{e \mapsto e'}{\text{If}(e, e_1, e_2) \mapsto \text{If}(e', e_1, e_2)} \quad \frac{}{\text{If}(\text{True}, e_1, e_2) \mapsto e_1} \quad \frac{}{\text{If}(\text{False}, e_1, e_2) \mapsto e_2} \\
 \\
 \frac{e_1 \mapsto e'_1}{\text{Let}(e_1, x.e_2) \mapsto \text{Let}(e'_1, x.e_2)} \quad \frac{e_1 \text{ done}}{\text{Let}(e_1, x.e_2) \mapsto e_2[e_1/x]}
 \end{array}$$

The rules for Plus are basically the same as in IMP. The rules for If are analogous to what we saw before; here, we have separate values True and False for the boolean type, and the If looks for those instead of treating 0 as false.

The rules for Let are a bit different than what we saw before with local variables; recall the rule

$$\frac{a \notin \text{dom}(\mu)}{\mu \mid \text{LocalVar}(x.s) \mapsto \mu, (a, \dots) \mid s[a/x]}$$

The main difference is that LocalVar declares a *mutable* variable, meaning that the variable stands for an address which can be set and get (e.g. `{var x; x := (get(x) + 1);}`). On the other hand, Let declares an *immutable variable*, which stands for the value of the expression e_1 . Formally, the operational semantics for LocalVar substitutes a memory address for the variable, while Let substitutes an expression for a variable. (Another difference is the presence/absence of e_1 : It makes sense not to provide an initial value for the mutable variable, because you can set it later. It wouldn't make sense not to provide an initial value for an immutable variable, because an immutable variable only ever has its initial value.)

We need to check that substituting an expression for a variable can be defined; the definition is basically the same as substituting an address:

$$\begin{aligned} x[e/x] &= e \\ y[e/x] &= y \text{ if } y \neq x \\ \text{Num}(k)[e/x] &= \text{Num}(k) \\ \text{Plus}(e_1, e_2)[e/x] &= \text{Plus}(e_1[e/x], e_2[e/x]) \\ \text{True}[e/x] &= \text{True} \\ \text{False}[e/x] &= \text{False} \\ \text{If}(e_0, e_1, e_2)[e/x] &= \text{If}(e_0[e/x], e_1[e/x], e_2[e/x]) \\ \text{Let}(e_1, y.e_2)[e/x] &= \text{Let}(e_1[e/x], y.e_2[e/x]) \text{ if } x \neq y \end{aligned}$$

In the final case, because we can always rename y to a different variable name, we assume that x is not equal to y . In an implementation, we could either rename y to some new variable, or stop the substitution from going into e_2 like we did before.

Here is a type system for this language:

$$\begin{array}{c} \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad \frac{}{\Gamma \vdash \text{Num}(k) : \text{int}} \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash \text{Plus}(e_1, e_2) : \text{int}} \\[10pt] \frac{}{\Gamma \vdash \text{True} : \text{bool}} \quad \frac{}{\Gamma \vdash \text{False} : \text{bool}} \quad \frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{If}(e, e_1, e_2) : \tau} \\[10pt] \frac{\Gamma \vdash e : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{Let}(e_1, x.e_2) : \tau_2} \end{array}$$

In the If rule, the two branches must have the same type. This is because there are no run-time type tests in this language — types are *static* or *compile-time* thing. (We will come back to the kind of run-time “type” tests that you might be used to from e.g. Python later on in the course.) In the Let rule, the idea is that x will stand for the value of e_1 , so whatever type e_1 has is the type we should assume x has while type-checking e_2 . Since the Let will eventually step to $e_2[e_1/x]$, the type of e_2 should be the same as the result type of the Let itself.

4 Progress

As we discussed above, there are some expressions e that, according to the above operational semantics, have undefined behavior. For example, $\text{Plus}(\text{True}, 1)$ is neither done nor does it take any step, and similarly for $\text{If}(0, \dots, \dots)$ since we have restricted the if's condition to be either true or false (not a number like in IMP).

However, the idea is that this is OK because we will first type-check a program and only then run it, so the operational semantics only needs to define the behavior of *well-typed* programs.

We can verify this with a *progress theorem*:

Theorem 1 (Progress). *If $\cdot \vdash e : \tau$ then either e is done or there exists an e' such that $e \mapsto e'$.*

This says that every *closed* well-typed program is either done or takes some step. A program is closed when it type-checks in the empty context $\cdot$, which checks that it does not use any variables from the ambient scope; the program might still define some local variables with `Let`.

The way we prove this is by induction on the typing rules, which means we have one case for each typing rule, and we check in each case that the expression is either done or takes a step.

Proof. • Case for

$$\frac{x : \tau \in \cdot}{\cdot \vdash x : \tau}$$

This case is impossible because there are no variables in scope when the context is empty.

• Case for

$$\frac{}{\cdot \vdash \text{Num}(k) : \text{int}}$$

In this case, $\text{Num}(k)$ is done (which implies that it is either done or takes a step).

• Case for

$$\frac{\cdot \vdash e_1 : \text{int} \quad \cdot \vdash e_2 : \text{int}}{\cdot \vdash \text{Plus}(e_1, e_2) : \text{int}}$$

Because this rule has two premises, stating that $\cdot \vdash e_1 : \text{int}$ and $\cdot \vdash e_2 : \text{int}$, we get two inductive hypotheses:

1. Either e_1 is done or there exists some e'_1 such that $e_1 \mapsto e'_1$
2. Either e_2 is done or there exists some e'_2 such that $e_2 \mapsto e'_2$

We will need to distinguish three cases on these possibilities:

- First, suppose the first inductive hypothesis says that $e_1 \mapsto e'_1$. Then

$$\frac{e_1 \mapsto e'_1}{\text{Plus}(e_1, e_2) \mapsto \text{Plus}(e'_1, e_2)}$$

so $\text{Plus}(e_1, e_2)$ takes a step, and therefore is either done or takes a step.

- Second, suppose the first inductive hypothesis says that e_1 is done. Then we distinguish cases on the second inductive hypothesis:

- * If the second inductive hypothesis says that $e_2 \mapsto e'_2$, then we have

$$\frac{e_2 \mapsto e'_2 \quad e_1 \text{ done}}{\text{Plus}(e_1, e_2) \mapsto \text{Plus}(e_1, e'_2)}$$

so $\text{Plus}(e_1, e_2)$ takes a step, and therefore is either done or takes a step.

- * If the second inductive hypothesis says that e_2 is done, then both e_1 and e_2 are done. Recall that the input typing derivation includes subderivations of $\cdot \vdash e_1 : \text{int}$ and $\cdot \vdash e_2 : \text{int}$. So the question is, if an expression is both done and has type int , what do we know about it? It turns out that we can prove a *canonical forms* principle:

If $\cdot \vdash e : \text{int}$ and e done then there exists a k such that $e = \text{Num}(k)$.

Proof: There are only three forms of done expressions, True , False , and $\text{Num}(k)$. Of those, there is a typing rule that says that $\text{Num}(k) : \text{int}$, but there is no typing rule saying that $\text{True} : \text{int}$ or $\text{False} : \text{int}$ (there is only one typing rule that applies to each of these, and they say that $\text{True} : \text{bool}$).

By the canonical forms principle, $e_1 = \text{Num}(k_1)$ and $e_2 = \text{Num}(k_2)$, so we need to show that $\text{Plus}(\text{Num}(k_1), \text{Num}(k_2))$ is either done or takes a step. But we have

$$\text{Plus}(\text{Num}(k_1), \text{Num}(k_2)) \mapsto \text{Num}(k_1 + k_2)$$

so it takes a step.

- There are four more cases for True , False , If , and Let , but we will leave those for the homework. □

Hopefully this proof looks familiar: it is analogous to your `progress_exp` function on the homework! That function can be seen as an implementation of the progress principle, returning `Done` when the above proof concludes e Done, and returning `Stepped e'` when the above proof concludes $e \mapsto e'$.

5 Preservation

The progress theorem says that every well-typed program is either done or takes a single step. To fully run a program we would like to iterate progress to obtain a chain of steps

$$e \mapsto e_1 \mapsto e_2 \mapsto \dots \mapsto e_n$$

where e_n is done (or continuing on forever in the case of an infinite loop). However, to prove that such a chain exists, we need to know that the e_1 such that $e \mapsto e_1$ is produced by the progress theorem is well-typed, so that we can feed e_1 into the progress theorem to get e_2 , etc.

This is called *preservation* — the type of a program is preserved by stepping.

Theorem 2. *Preservation* If $e \mapsto e'$ and $\cdot \vdash e : \tau$ then $\cdot \vdash e' : \tau$

This one is proved by induction on stepping, which means we have one case for each step rule.

Proof. • Case for

$$\frac{e_1 \mapsto e'_1}{\text{Plus}(e_1, e_2) \mapsto \text{Plus}(e'_1, e_2)}$$

Assume $\cdot \vdash \text{Plus}(e_1, e_2) : \tau$. Since there is only one rule that can type check a Plus, we must have that τ is int and $\cdot \vdash e_1 : \text{int}$ and $\cdot \vdash e_2 : \text{int}$. (This step is called *inversion*, and could be broken out as a lemma. However, when the type system is structured as exactly one rule for each form of syntax, the inversion principle is true.) Thus, we need to show that $\text{Plus}(e'_1, e_2) : \text{int}$.

By the inductive hypothesis, since $e_1 \mapsto e'_1$ and $\cdot \vdash e_1 : \text{int}$, we have that $\cdot \vdash e'_1 : \text{int}$. So we can reapply the typing rule for plus to conclude

$$\frac{\cdot \vdash e'_1 : \text{int} \quad \cdot \vdash e_2 : \text{int}}{\cdot \vdash \text{Plus}(e'_1, e_2) : \text{int}}$$

- Case for

$$\frac{e_2 \mapsto e'_2 \text{ done}}{\text{Plus}(e_1, e_2) \mapsto \text{Plus}(e_1, e'_2)}$$

This case is very similar to the above. Assume $\cdot \vdash \text{Plus}(e_1, e_2) : \tau$. By inversion, we must have that τ is int and $\cdot \vdash e_1 : \text{int}$ and $\cdot \vdash e_2 : \text{int}$. Thus, we need to show that $\text{Plus}(e_1, e'_2) : \text{int}$.

By the inductive hypothesis, since $e_2 \mapsto e'_2$ and $\cdot \vdash e_2 : \text{int}$, we have that $\cdot \vdash e'_2 : \text{int}$. So we can reapply the typing rule for plus to conclude

$$\frac{\cdot \vdash e_1 : \text{int} \quad \cdot \vdash e'_2 : \text{int}}{\cdot \vdash \text{Plus}(e_1, e'_2) : \text{int}}$$

- Case for

$$\frac{}{\text{Plus}(\text{Num}(k_1), \text{Num}(k_2)) \mapsto \text{Num}(k_1 + k_2)}$$

Assume $\cdot \vdash \text{Plus}(e_1, e_2) : \tau$. By inversion, we must have that τ is int, so we need to show that $\text{Num}(k_1 + k_2) : \text{int}$, which it does.

- Case for

$$\frac{e_1 \mapsto e'_1}{\text{Let}(e_1, x.e_2) \mapsto \text{Let}(e'_1, x.e_2)}$$

Assume $\cdot \vdash \text{Let}(e_1, x.e_2) : \tau$. The inversion principle for Let is that $\cdot \vdash e_1 : \tau_1$ and $x : \tau_1 \vdash e_2 : \tau$ — the only rule that type checks a let has these as premises.

By the inductive hypothesis, since $e_1 \mapsto e'_1$ and $\cdot \vdash e_1 : \tau_1$, we have $\cdot \vdash e'_1 : \tau_1$. So we can conclude

$$\frac{\cdot \vdash e'_1 : \tau_1 \quad x : \tau_1 \vdash e_2 : \tau}{\cdot \vdash \text{Let}(e'_1, x.e_2) : \tau}$$

-

$$\frac{e_1 \text{ done}}{\text{Let}(e_1, x.e_2) \mapsto e_2[e_1/x]}$$

Assume $\cdot \vdash \text{Let}(e_1, x.e_2) : \tau$. By inversion $\cdot \vdash e_1 : \tau_1$ and $x : \tau_1 \vdash e_2 : \tau$.

We need to show that $\cdot \vdash e_2[e_1/x] : \tau$. This holds because of the *substitution lemma*, which says that if you substitute an expression e_1 for a variable x whose type τ_1 matches the type of e_1 into a well-typed expression, then the result of the substitution is well-typed. We will look at the proof of the substitution lemma next time.

- There are more cases for If but we will leave those for the homework.

□

Lemma 3 (Substitution). *If $\cdot \vdash e_1 : \tau_1$ and $x : \tau_1 \vdash e_2 : \tau$ then $\cdot \vdash e_2[e_1/x] : \tau$.*