Typechecking Basic Exprssions

Kyle Dewey

1 Introduction

This document covers the process of *typechecking*, that is, of determining if an input program is well-typed or ill-typed. This assumes the reader is familiar with context-free grammars and abstract syntax trees. This is intended to be a quick introduction for someone who needs to implement a compiler fast, as opposed to a full introduction to type theory.

1.1 Expressions and Statements

In most programming languages, programs are composed primarily of *expressions* and *statements*. Expressions produce a value, whereas statements do not produce values. Usually, statements are useful for some sort of effect they have on the code. For example, 1, $1 + 2$, and $x * 5$ are all expressions; these evaluate down to some value. In contrast, `int x = 5;` is a statement; this has the effect of declaring a variable and initializing it with the value of 5, and does not produce a value overall.

If this still isn't clear, ask yourself: "can I assign this into a variable?" If yes, it's an expression, if not, it's a statement (for our purposes, anyway). For example, dovetailing off of the previous examples, `myVar = 1`, `myVar = 1 + 2`, and `myVar = x * 5` all make sense, whereas `myVar = int x = 5;` does not.

Note that statements can contain expressions within them. Using the prior examples, 5 is an expression, but `int x = 5;` is a statement. Some languages also permit statements to be embedded in expressions. Exactly what is permissible is defined by the language's grammar; most languages will have production rules for expressions and statements.

1.2 Types

Most languages have the concept of *types*. Types define the kinds of values which are possible in the language, along with the operations which are permissable on specific values. For example, consider the following Java code:

```
int x = 1;
int y = 2;
int z = x + y;
```

Java knows that `x`, `y`, and `z` are all variables of type `int`, and that `+` is a valid operation that can be performed with two values of type `int` (specifically `x` and `y` in the example above). Moreover, the result of the `+` operation is `int`.

1.2.1 Statically-Typed Languages

Java is an example of a *statically-typed language*. This means that the types of all variables are known at compile time. In statically-typed languages, programs with type errors (i.e., types are used in an inconsistent manner) are rejected (i.e., they fail to compile).

1.2.2 Dynamically-Typed Languages

Python (older variants) and JavaScript are examples of *dynamically-typed languages*. This means that types are associated with *values*, not variables. This is far less restrictive than with statically-typed languages. For example, the following is legal Python code:

```
x = 7
x = "foo"
```

With this program, while `7` is of type `int` and `''foo''` is of type `String`, `x` itself has no fixed type associated with it; `x`'s type depends on whichever value it is holding at the moment. This program cannot be written in any statically-typed language, because `x`'s type isn't permitted to change in a statically-typed language.

Dynamically-typed languages are less restrictive than statically-typed languages, but this comes with a cost. For one, performance-wise, generally the more information the compiler has at compile time, the better it can optimize your code. For instance, if the compiler knows that `x` is an `int`, and `int` values are 64 bits on your platform, then it can allocate exactly 32 bits for your variable. However, if the compiler doesn't know the type of `x`, it will usually have to allocate space to hold a 'pointer to a 'box", where the box itself holds the value. This means we'd need 64 bits for the pointer (assuming a 64 bit platform), which itself would point to a box. The box would need to be at least 64 bits large for the integer, and is usually a bit larger. Moreover, we add an extra level of indirection for every access to `x`; to get the integer out, we no longer just look up `x`, we also have to dereference `x` (two memory lookups as opposed to one).

Beyond performance, dynamically-typed languages can be overly permissive, to the point where they make life difficult. If a type error is possible in a dynamically-typed program, then we must find just the right input that will trigger it. A bug is present, but it's hidden from us.

1.3 Static Typechecking

For our purposes, we will assume we're working with a statically-typed language, and are typechecking the program at compile time. The process of typechecking is usually phrased as a series of rules which explain how to derive what the type of a given program is. For example, with $1 + 3$, we would need (at least) two rules working in conjunction:

- The type of any integer (e.g., 1 and 3) is `int`
- The type of $e_1 + e_2$, where e_1 is `int` and e_2 is `int`, is `int`

Rules can be specified in natural languages (as English is mostly used above), but this tends to be problematic. Natural languages tend to be both verbose and imprecise, leading to lengthy, unclear descriptions. Instead, it is usually preferable to use *inference rules*, which can be seen as a way of writing an unambiguous specification using math. The next section will introduce inference rules via example.

2 Complete Example

The remainder of this document will show typechecking rules for a relatively simple language. While this language is simple, this will end up covering all the basics we'll need to build things up with.

2.1 Language Used

Rather than start with an existing language, we'll instead define our own language. Existing languages tend to be very complex, so they don't work well when we're learning the basics.

We will define a statically-typed, expression-based language. By saying the language is expression-based, this means that *everything* in the language is an expression, no exceptions. This makes life easier when defining typing rules (i.e., the inference rules saying how to derive a program's type). Before jumping to this language's grammar, we'll first see some example programs, divided up by feature.

2.1.1 Booleans

Our language supports booleans. Specifically, `true` is a program that returns the boolean value `true`, `false` is a program that returns the boolean value `false`, and the `&&` operation returns the boolean conjunction of two boolean subexpressions. For example, each one of the following lines is a complete program:

```
true
false
true && false
true && (false && true)
```

Boolean expressions have type `bool`; each of the above programs is of type `bool`.

2.1.2 Integer Arithmetic

Our language supports integer arithmetic. This includes all possible integers (e.g., `1`, `42`, `118`) as expressions which evaluate themselves, as well as the `+` and `<` operations with their usual meanings. For example, each one of the following lines is a complete program:

```
47
8 + 2
1 + 2 + 3
1 + (2 + 3)
(7 + 8) < 27
```

Integer expressions have type `int`; each of the above programs is of type `int`.

2.1.3 Variables and Assignment

The last key feature is variables and variable assignment. Variables can be declared using `let`, as shown below:

```
let x: int = 7 in
  x + x
```

Note that the entire above program is *a single expression*; by making this one expression, we have no need to separately introduce statements. In the above program, `let` specifically does the following, in order:

- Declares the new variable `x` to be of type `int`.
- Initializes `x` to hold `7`.
- Executes `x + x`, where `x` is in scope from the prior part.

With all this in mind, the above program evaluates to `14`.

Assignment looks similar to `let`. For assignment to work, it needs a variable to already be in scope. This is shown in the program below:

```
let x: int = 8 in
  assign x = 9 in
  x
```

The above program does the following, in order:

- Declares `x` to be of type `int`, and initializes `x` to `8`
- Assigns `9` to `x`
- Evaluates to whatever the current value of `x` is

With the above in mind, the above program evaluates to `9`.

Unlike `let`, `assign` does not introduce a new scope. This is because `assign` cannot introduce a new variable, only modify an existing variable. We attempt to show this visually with whitespace above; the last `x` is at the same level as `assign`, as opposed to being indented in.

2.1.4 Putting it All Together: Formal Abstract Syntax

Now that we have a sense of the language informally, we can define it a bit more formally. To that end, we will formally define its *abstract* syntax via a BNF grammar. With an abstract syntax, we don't care about specifics which are relevant only to parsing (e.g., left recursion is ok, operator precedence is irrelevant, etc.); our goal with abstract syntax is to define what legal abstract syntax trees look like, not the specific construction of abstract syntax trees from raw input (i.e., parsing). Oftentimes, languages have both a concrete syntax and abstract syntax defined, where the concrete syntax is specifically for the parser. Once the parser is through with its job, we can work with the much simpler abstract syntax instead. In our case, since we are only considered with typechecking, we skip over the concrete syntax entirely; typechecking follows parsing, so we can effectively skip over parsing-related concerns.

The abstract syntax for this language is below:

$$x \in \textit{Variable} \qquad i \in \textit{Integer}$$

$$\begin{aligned}
\tau \in \textit{Type} &::= \textsf{int} \mid \textsf{bool} \\
e \in \textit{Exp} &::= x \mid i \mid \textsf{true} \mid \textsf{false} \mid e_1 \mathbin{\&\&} e_2 \mid e_1 + e_2 \mid e_1 < e_2 \\
&\mid \textsf{let } x : \tau = e_1 \textsf{ in } e_2 \\
&\mid \textsf{assign } x = e_1 \textsf{ in } e_2
\end{aligned}$$

These BNF rules correspond to the sort of informal examples we've seen already. Some points about the notation are below:

- As usual, ::= introduces a new production rule. The specific notation before it (e.g., $e \in \textit{Exp}$) defines the name of the production rule (*Exp*), as well as a *metavariable* which stands-in for the production rule (e). The idea is that we can use a metavariable to refer to the production again.
- The specific metavariable used says what kind of production we are referring to. For example, τ means we are referring to a type (*Type*), and e says we are referring to an expression (*Exp*). Subscripts allow us to distinguish between different expansions of a production; for example, if we have e_1 and e_2 in some context, this means that e_1 and e_2 are permitted to be two different expressions.
- For items defined without production rules (e.g., $x \in \textit{Variable}$), we effectively leave what these components are abstract, allowing the parser to fill in the details. From an implementation standpoint, these components usually refer to individual tokens. They are effectively primitive values.
- τ and e are conventionally used as metavariables for types and expressions, respectively.

2.2 Type System Without Variables

Now that we have formally defined the language, we can formally define the language's *type system*. A type system is a set of rules which say exactly how to determine what the type of a given program is. We incrementally define the type system below. We intentionally delay a discussion of how to handle variables to the next section; this strictly adds more complexity.

2.2.1 Direct Integer and Boolean Values

We start our definition with integers and booleans which are directly specified. Specifically, we want to say:

- Any integer i is of type `int`
- `true` and `false` are each of type `bool`

We encode this information with the following rules:

$$\frac{}{i : \textsf{int}}\ (\textsc{integer}) \qquad \frac{}{\textsf{true} : \textsf{bool}}\ (\textsc{true}) \qquad \frac{}{\textsf{false} : \textsf{bool}}\ (\textsc{false})$$

Each of these rules handles a different part of the above description. The part in parenthesis (e.g., (INTEGER)) gives a human-readable name to the rule; this is not strictly necessary for our purposes. Each use of : says that some expression is of a certain type (e.g., i : **int** says that any integer i is of type **int**). The line over each rule is also important. Because there is nothing written above this line, this says that each of these rules is an *axiom*; each is always true, no matter what. We'll see a case where something is written above the line shortly.

2.2.2 Integer and Boolean Operations

We now define the operations on integer and booleans. In plain English, we want to say:

- e_1 && e_2 is of type `bool`, as long as e_1 and e_2 are both of type `bool`
- $e_1 + e_2$ is of type `int`, as long as e_1 and e_2 are both of type `int`

- $e_1 < e_2$ is of type `bool`, as long as e_1 and e_2 are both of type `int`

Rules encoding the above information follow:

$$\frac{e_1 : \textbf{bool} \quad e_2 : \textbf{bool}}{e_1 \;\&\&\; e_2 : \textbf{bool}}\ (\text{AND}) \qquad \frac{e_1 : \textbf{int} \quad e_2 : \textbf{int}}{e_1 + e_2 : \textbf{int}}\ (\text{PLUS}) \qquad \frac{e_1 : \textbf{int} \quad e_2 : \textbf{int}}{e_1 < e_2 : \textbf{bool}}\ (\text{LESS-THAN})$$

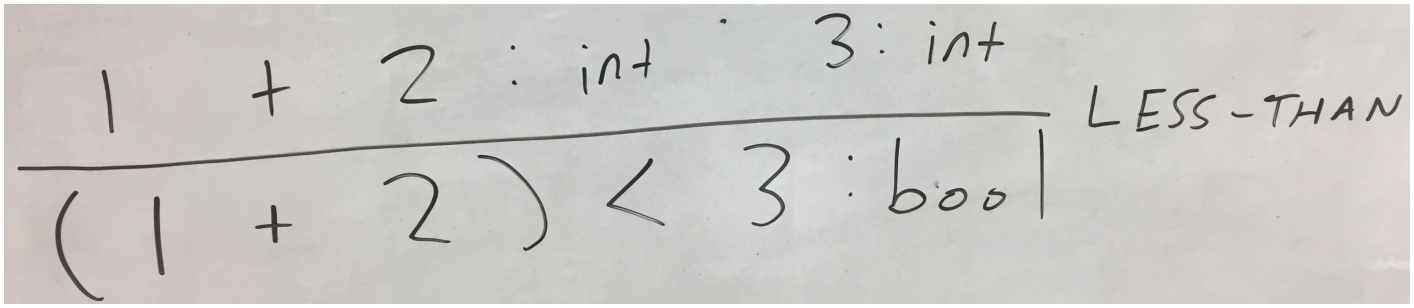
Note that the lines over the rules now have *premises* above them. These rules now apply only conditionally; they are no longer axioms. For example, the rule handling $e_1 + e_2$ now only applies if e_1 and e_2 are both of type `int`.

2.2.3 Typechecking

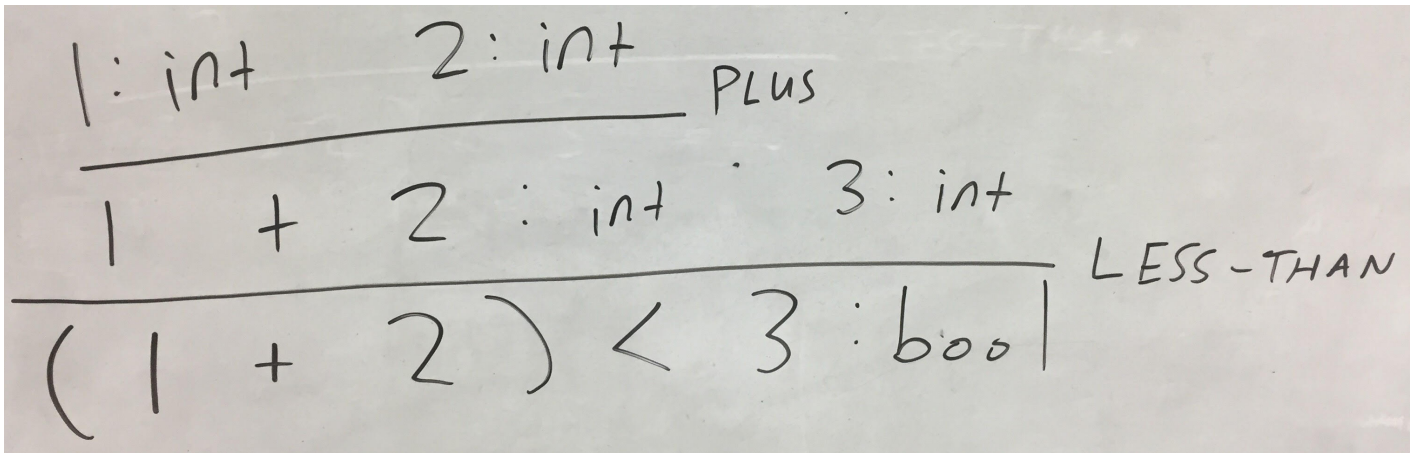
We can apply these rules directly to typechecking, and even perform the process on paper. Specifically, we start with a program, and then work through it from the bottom to the top. To illustrate this process, let's consider the following program:

```
(1 + 2) < 3
```

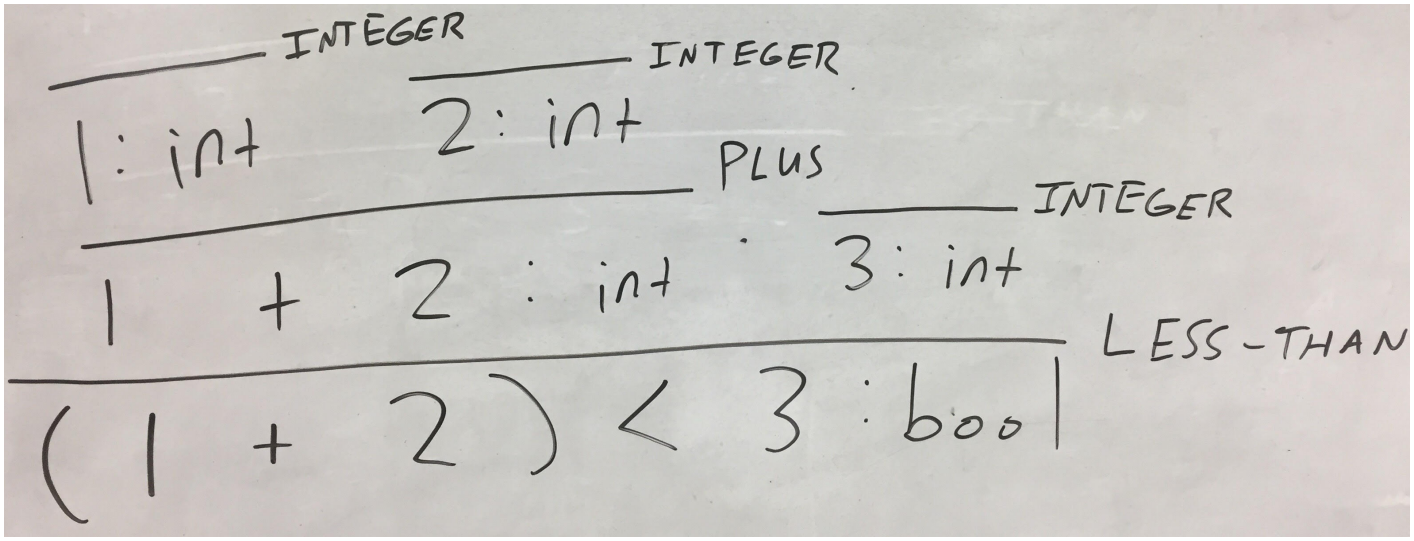
Looking at this program, we can immediately try to apply the LESS-THAN rule. This is shown below:



Note, however, we're not done: we now need to check the premises on `less-than`. At this point, we can apply the PLUS rule to handle $1 + 2$, as shown below:



As for 1, 2, and 3, we can handle these all with INTEGER, like so:

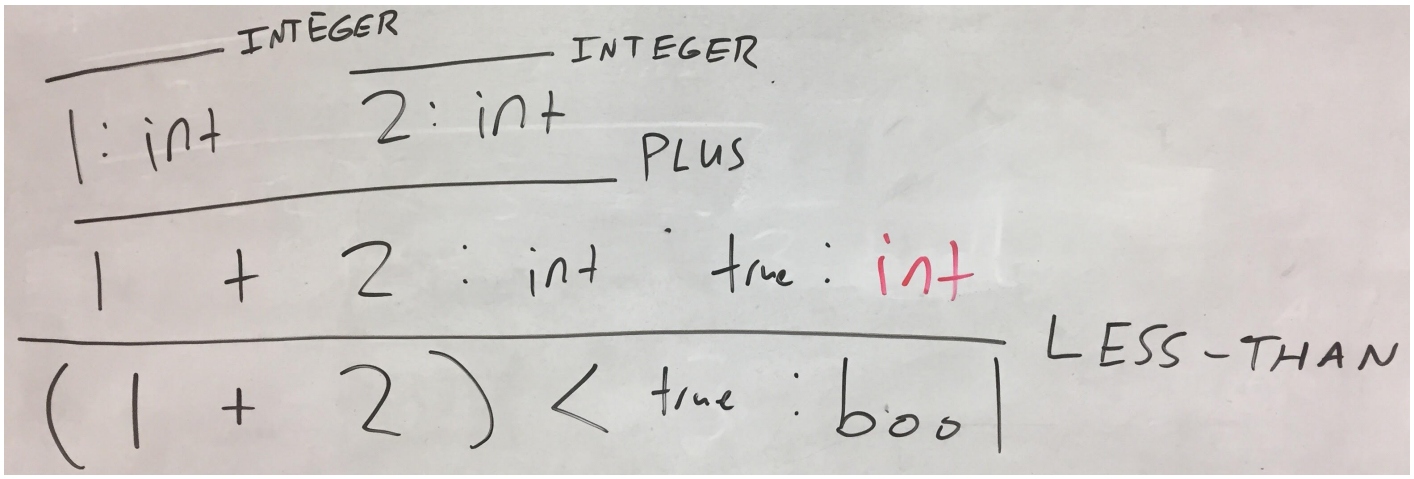


At this point, there is no further work to do; we have recursively checked all the necessary premises, and everything in the image starts with a line. We now know that the type of this program is `bool`.

Sometimes, we cannot deduce the type of a program. This usually happens because an input program is *ill-typed*; that is, it has no associated type, and there is at least one type error somewhere. To see what happens in the event of an ill-typed program, let's try to apply this same process to the following ill-typed program:

```
(1 + 2) < true
```

This starts off the same as last time, up through the application of the INTEGER rule:



However, this time around, we hit a problem, indicated by the spot in red. The TRUE rule is the only rule that directly applies to `true`, *but* TRUE cannot be applied here because we expect the type to be `int`, not `bool`. That is, both the expression itself and the type need to match up with the rule, otherwise the rule cannot be applied. Here we get stuck - we need to apply *something* here (the tops of all premises need to have a line, so we're not done yet), but there is nothing we can apply. This sort of getting stuck happens with ill-typed programs; more formally, the expression `(1 + 2) < true` has no type.

2.2.4 Code Glimpse

So far, the typing rules can be viewed as a way of collectively defining a Java function with the following signature:

```
public Type typeof(Exp e) throws IllTypedException
```

. . . where:

- `Type` is a class representing a type
- `Exp` is a class representing an expression
- `IllTypedException` is an exception thrown if we discover that `e` is ill-typed.

All the rules collectively define the implementation of `typeof`.

2.3 Type System With Variables

At this point, we've handled all the rules that do not involve variables. To add variables, we'll need to keep track of which variables are in scope, along with the types of those variables. The usual way to do this is by adding a *type environment*, which maps variables in scopes to their corresponding types. Formally, we can write this as follows:

$$\Gamma \in \mathit{TypeEnv} = \mathit{Variable} \to \mathit{Type}$$

The above definition states that metavariable Γ represents a mapping from *Variable* to *Type*. That is, if we give a Γ a *Variable*, then it will give us back a *Type*. That said, Γ will only give us back a type if there is such a variable in the domain of Γ.

To make this more concrete, we will show a modified version of `typeof` as a Java signature which operates with a Γ:

```
public Type typeof(Exp e, Map<Variable, Type> gamma) throws IllTypedException
```

As shown, the type environment is literally just a plain old map data structure (or dictionary, if you prefer), where the keys are variables and the values are types. We now pass the type environment to `typeof`, along with the expression.

There is, however, one slight distinction between Γ and a Java map: Γ is an *immutable* map, whereas `Map` is generally *mutable*. The difference lies in what happens when we add a new key/value pair to the map. In the math, adding a new key/value pair returns a *new* map, which is the old map with the new key/value pair. The original map is unchanged. In contrast, in Java, adding a new key/value pair generally modifies the original map. For what it's worth, many languages have libraries supporting the same sort of immutable data structures in code, it's just that Java doesn't have this out of the box.

Towards handling variables, we'll need to update our rules to pass along Γ. We do this below, modifying the same rules we've seen so far:

$$\frac{}{\Gamma \vdash i : \textsf{int}}\ (\textsc{integer}) \qquad \frac{}{\Gamma \vdash \textsf{true} : \textsf{bool}}\ (\textsc{true}) \qquad \frac{}{\Gamma \vdash \textsf{false} : \textsf{bool}}\ (\textsc{false})$$

$$\frac{\Gamma \vdash e_1 : \textsf{bool} \quad \Gamma \vdash e_2 : \textsf{bool}}{\Gamma \vdash e_1 \mathbin{\&\&} e_2 : \textsf{bool}}\ (\textsc{and}) \qquad \frac{\Gamma \vdash e_1 : \textsf{int} \quad \Gamma \vdash e_2 : \textsf{int}}{\Gamma \vdash e_1 + e_2 : \textsf{int}}\ (\textsc{plus}) \qquad \frac{\Gamma \vdash e_1 : \textsf{int} \quad \Gamma \vdash e_2 : \textsf{int}}{\Gamma \vdash e_1 < e_2 : \textsf{bool}}\ (\textsc{less-than})$$

Let's start discussion of these rules with INTEGER. The notation $\Gamma \vdash i : \textsf{int}$ says that the input type environment is Γ and the input expression is i. $\vdash$ acts as a separator of the type environment and expression inputs, just like : separates the input expression from the output type. Moving our discussion onto AND, we can see that each recursive use of our typing rules similarly needs to have a type environment passed along as a parameter; that is, typechecking e_1 and e_2 now need Γ as a parameter.

Note that none of these rules manipulate Γ directly; they merely take Γ and pass it along. Considering our language, this should make sense; none of these rules directly involve variables, and Γ is only needed when working with variables. As such, we expect that these rules *not* to touch Γ.

With that, let's introduce the rules that *do* touch Γ, starting with variables:

$$\frac{x \in \texttt{dom}(\Gamma) \quad \tau = \Gamma[x]}{\Gamma \vdash x : \tau}\ (\textsc{var})$$

The notation $x \in \texttt{dom}(\Gamma)$ checks if x is in the domain (`dom`) of Γ (i.e., x is contained in the type environment). If not, the VAR rule does not apply. The notation $\Gamma[x]$ looks up the key associated with x in Γ. The result of this lookup is bound to metavariable τ. In this context, τ is some type, and we are permitted to introduce as many metavariables as we wish. We end up saying that the type of x must be whatever τ is.

In plain English, the above rule:

- Checks to see if x is in the type environment (in scope). If not, then this rule doesn't apply, and the program will be considered ill-typed.

- Returns whatever type is associated with x in Γ, performing a map lookup to check this.

From here, we introduce rules handling `let` and `assign`, shown below:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \textsf{let } x : \tau_1 = e_1 \textsf{ in } e_2 : \tau_2} \text{ (LET)} \qquad \frac{\begin{array}{c} x \in \texttt{dom}(\Gamma) \quad \tau_1 = \Gamma[x] \\ \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \end{array}}{\Gamma \vdash \textsf{assign } x = e_1 \textsf{ in } e_2 : \tau_2} \text{ (ASSIGN)}$$

Let's first look at LET. The premises of LET say the following:

- $\Gamma \vdash e_1 : \tau_1$ states that e_1 must be of type τ_1 underneath Γ. Notably, τ_1 must be the *same type* as the user annotated x to be in the program.
- The notation $\Gamma[x \mapsto \tau_1]$ adds a key/value pair to the map. Specifically, x is the key, and τ_1 is the value. If x is already a key in Γ, then this will effectively overwrite x's current value with τ_1 (that is, the returned type environment will map x to τ_1, instead of whatever x previously mapped to).
- With the prior point in mind, $\Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2$ states to put x in scope, associate it with type τ_1, and determine the type of e_2. The type of e_2 is bound to metavariable τ_2.
- Finally, the type of a whole `let` expression is τ_2, the type of e_2 with x in scope.

Given what we know so far, ASSIGN should be straightforward. This specifically states that:

- Whatever variable being assigned to needs to already be in scope ($x \in \texttt{dom}(\Gamma)$)
- The type of the value we assign to x (e_1) must be the same type as x ($\tau_1 = \Gamma[x]$ and $\Gamma \vdash e_1 : \tau_1$)
- The type of the overall assignment is whatever the type of e_2 is (τ_2)

3 All in One Place

We previously said that the advantage of this sort of mathematical formalism is conciseness and clarity. To illustrate this, the meat of everything we've discussed is repeated below, without the corresponding explanations. All the explanations are entirely redundant with this; in a formal setting, only the following needs to be shown.

3.1 Abstract Syntax

$$x \in \textit{Variable} \qquad i \in \textit{Integer}$$

$$\begin{aligned} \tau \in \textit{Type} &::= \textsf{int} \mid \textsf{bool} \\ e \in \textit{Exp} &::= x \mid i \mid \textsf{true} \mid \textsf{false} \mid e_1 \mathbin{\&\&} e_2 \mid e_1 + e_2 \mid e_1 < e_2 \\ &\quad \mid \textsf{let } x : \tau = e_1 \textsf{ in } e_2 \\ &\quad \mid \textsf{assign } x = e_1 \textsf{ in } e_2 \end{aligned}$$

3.2 Type System

$$\Gamma \in \textit{TypeEnv} = \textit{Variable} \to \textit{Type}$$

$$\frac{}{\Gamma \vdash i : \textsf{int}} \text{ (INTEGER)} \qquad \frac{}{\Gamma \vdash \textsf{true} : \textsf{bool}} \text{ (TRUE)} \qquad \frac{}{\Gamma \vdash \textsf{false} : \textsf{bool}} \text{ (FALSE)}$$

$$\frac{\Gamma \vdash e_1 : \textsf{bool} \quad \Gamma \vdash e_2 : \textsf{bool}}{\Gamma \vdash e_1 \mathbin{\&\&} e_2 : \textsf{bool}} \text{ (AND)} \qquad \frac{\Gamma \vdash e_1 : \textsf{int} \quad \Gamma \vdash e_2 : \textsf{int}}{\Gamma \vdash e_1 + e_2 : \textsf{int}} \text{ (PLUS)} \qquad \frac{\Gamma \vdash e_1 : \textsf{int} \quad \Gamma \vdash e_2 : \textsf{int}}{\Gamma \vdash e_1 < e_2 : \textsf{bool}} \text{ (LESS-THAN)}$$

$$\frac{x \in \texttt{dom}(\Gamma) \quad \tau = \Gamma[x]}{\Gamma \vdash x : \tau} \text{ (VAR)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \textsf{let } x : \tau_1 = e_1 \textsf{ in } e_2 : \tau_2} \text{ (LET)} \qquad \frac{\begin{array}{c} x \in \texttt{dom}(\Gamma) \quad \tau_1 = \Gamma[x] \\ \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \end{array}}{\Gamma \vdash \textsf{assign } x = e_1 \textsf{ in } e_2 : \tau_2} \text{ (ASSIGN)}$$