A Tutorial on Type Theory, Foundations of Programming Languages, and Formal Verification

John Altidor

1 Introduction

Type theory is a logical formalism used extensively in the study and design of programming languages to define the semantics and behavior of deductive systems. According to [17], "The central organizing principle of language design is the identification of language features with types. The theory of programming languages, therefore, reduces to the theory of types." Type theory ranges over several formal systems, but for this paper, type theory refers to the design, analysis and study of *type systems*. Due to the wide variety of usage of the phrase "type system", it lacks a standard definition. According to [14]: "A type system is a collection of rules that assign a property called a type to the various language constructs- such as variables, expressions, functions or modules- that a computer program is composed of." A type system formally defines many aspects of programming language such as in the following far-from-exhaustive list:

1. Which expressions are allowed in the language; specifically, what a well-formed or well-*typed* program in the language is. Vaguely, well-typed programs are programs that entail certain properties such as the absence of certain program errors. [19]

2. How program abstractions and components of large systems can be tied together.

3. How expressions in the language are evaluated. What is the order of evaluation. Are expressions evaluated eagerly (every expression in program is computed immediately) or lazily (expressions are not evaluated until the program requires their values).

Type systems model complex language features and enable one to prove properties about languages. Type systems are rigorous enough to be used as a language specification for a compiler writer to implement the language; the implementation of a programming language can follow from its type system.

A type system is specified by a set of inference rules that define a programming language. These inference rules are partitioned into two categories. Rules defining the types of the *terms* or expressions in the language are the *static semantics*. Static semantics *inductively* define a relation between expressions and types. *Dynamic semantics* or operational semantics inductively define how to evaluate expressions in the language. Specifically, dynamic semantics define a transition system between expressions, where the *values* that expressions evaluate to are the *final states* of the system.

Type systems are best explained with an example. The following sections define a programming language we coin as *MiniLang*. The *grammar* of *MiniLang* is defined in Section 2. Its static and dynamic semantics are given in Sections 3 and 4, respectively. We prove an important property, *type preservation*, for *MiniLang* in Section 5. We show to prove another important property, *progress*, by proving it for one type of expression in *MiniLang* in 6. The proof of progress for all other expressions is similar, so we skip those cases for brevity.

Proofs of language properties are important but long, tedious, and error prone because there are many cases to reason about. As a result, *proof assistants* have been developed for proving language theory. These software tools provide a language for writing properties and proofs of properties. These tools can find mistakes in proofs and verify proof correctness. We cover one proof assistant, Twelf [11], in Section 7. All of the Twelf code presented in this paper can be found in [15]. Section 8 concludes with a summary.

2 *MiniLang* Grammar

This section presents the *syntax* of *MiniLang*, a language of numbers and strings. The following notation may seem unusual because most programmers write programs in the *concrete syntax* of a language. The concrete syntax of language specifies how humans write programs in the language. Type systems are often written over the *abstract syntax* of the language to reflect that expressions in the language are *abstract syntax trees* (ASTs) or more simply called *terms*. ASTs are mathematical-like expressions that represent a composition of *nodes*. Each AST has the form:
$operator(operand_1, operand_2, \ldots, operand_n)$, where each operand is an AST, the operator is a root node of these ASTs, and $n \geq 0$. AST nodes are operators that take a specified number of operands. An operator can take in zero operands; in this case, the parentheses after the operator are typically not written. For instance, in the AST `add(6, 1)`, the node `6` could have been written as `6()`, and `add(6, 1)` is equivalent to `add(6(), 1())`. Only the abstract syntax is needed to reason about a language.

The abstract and concrete syntax of a language is defined with a *formal grammar* that specifies *production rules* on how AST nodes are constructed. Each production rule has the form "$A ::= B_1 \mid B_2 \mid \ldots \mid B_n$". A *non-terminal symbol* A denotes a set of ASTs specified by a production rule, where A is on the left-hand side of the $::=$. Each B_i denotes a category of AST nodes or a *terminal symbol* denoting a specific AST node. The production rule specifies that the A symbol denotes the category of nodes that is the union of nodes in $B_1, B_2, \ldots, B_n$. Figure 1 shows the grammar of *MiniLang*.

Category	*Item*		*Abstract*	*Concrete*
Expression	e	$::=$	x	x
		$\mid$	num[n]	n
		$\mid$	str[s]	's'
		$\mid$	+(e_1; e_2)	e_1 + e_2
		$\mid$	^(e_1; e_2)	e_1 ^ e_2
		$\mid$	let(x; e_1; e_2)	let x be e_1 in e_2

Figure 1: Grammar of *MiniLang*

The only non-terminal symbol in *MiniLang* is e, which denotes the set of ASTs that are expressions. Expressions can be the following:

1. Numbers: num[n], where n is a sequence of digits.

2. Strings: str[s], where s is a sequence of characters.

3. Variable names: x

4. The sum of two subexpressions: +(e_1, e_2).

5. The string concatenation of two subexpressions: ^(e_1, e_2).

6. A let expression where the subexpression e_2 is evaluated in a context that has variable x bound to the value of e_1.

Example expressions are shown in Figure 2. For the remaining sections in this paper, only the abstract syntax will be shown.

Abstract	*Concrete*
`+(num[5]; +(num[4]; num[3]))`	`5 + 4 + 3`
`^(str[john]; ^(x; str[doe]))`	`'john' ^ x ^ 'doe'`
`let(hours; num[24]; +(hours; num[33])`	`let hours be 24 in hours+33`

Figure 2: Example *MiniLang* Expressions

3 Static Semantics

Although grammars only allow a limited set of ASTs, a language may want to filter out additional ASTs allowed by the grammar that are not well-formed or "make sense" according to the language specification. For *MiniLang* we will formally define the addition of two numbers and the concatenation of two strings. On the contrary, we will not define the addition between a number and a string and likewise for concatenation. We say such expressions are *ill-defined* or *ill-typed* and are not considered part of the *MiniLang* language. According to this specification, the following two expressions are ill-typed even though they are in the grammar of *MiniLang*.

1. `+(num[4], str[doe])`
2. `let(daysPerWeek, str[seven], +(num[1], daysPerWeek))`

Filtering out ill-typed ASTs is not always possible with only the grammar specification such as the second AST above. The reason is because the grammar is *context-free*, and determining that the second AST is ill-typed requires a *context-sensitive* analysis. Programming languages typically specify their syntax with context-free grammar because they are conceptually and computationally easier to parse than context-sensitive grammars. The reason why is beyond the scope of this paper.

This filtering process is performed by *type checking* the AST nodes. Type checking tries to derive a type to each AST node using the inference rules of the *static semantics* of a language. If no such type can be assigned to a node, the node is not considered to be well-typed (well-formed) and compilers would flag it as an error in the program.

Inferences rules have the following form:

$$\frac{\overbrace{J_1 \quad J_2 \quad \ldots \quad J_n}^{premises}}{\underbrace{J}_{conclusion}} \texttt{ Rule Label}$$

Each J_i is a proposition or *judgment*. If all of the judgments of the premise (J_i's) are true, then the conclusion judgment J is true. Rules with no premises are axioms because the conclusion is true under any conditions. Judgments asserting the type of an AST node typically have the form $\Gamma \vdash e \ : \ \tau$ saying that node e has type τ under the typing context Γ, where Γ is a function mapping variable names to types.

Consider type checking *MiniLang*. To discriminate between expressions that are numbers and strings, we define two types: `num` and `str`.

Figure 3 gives the typing rules for *MiniLang*. Rule `T.1` says that every number has type `num`. Rule `T.2` says that every string has type `str`. Rule `T.3` says that if the typing context maps a variable x to type τ, then in that context, x has type τ. Rule `T.4` says that the addition of two subexpressions that have type `num` is also a `num`. Rule `T.5` says that the concatenation of two subexpressions that have type `str` is also a `str`. Rule `T.6` is the more complicated rule that uses multiple typing contexts. It says that if e_1 (which will be the value of x in e_2) in context Γ has type τ_1 and if under the context of Γ *extended* with the mapping (x, τ_1) that e_2 has type τ_2, then the entire `let` expression has type τ_2.

Figure 4 shows how typing rules are applied to derive the type of an expression. Figure 5 shows how an ill-typed expression is discovered.

$$\frac{}{\Gamma \vdash \texttt{num}[n]\texttt{:}\ \ \texttt{num}}\ \texttt{T.1} \qquad \frac{}{\Gamma \vdash \texttt{str}[s]\texttt{:}\ \ \texttt{str}}\ \texttt{T.2} \qquad \frac{(x,\tau)\in\Gamma}{\Gamma\vdash x:\tau}\ \texttt{T.3}$$

$$\frac{\Gamma \vdash e_1\texttt{:}\ \ \texttt{num} \quad \Gamma\vdash e_2\texttt{:}\ \ \texttt{num}}{\Gamma\vdash \texttt{+(}e_1\texttt{;}\ e_2\texttt{):}\ \ \texttt{num}}\ \texttt{T.4} \qquad \frac{\Gamma \vdash e_1\texttt{:}\ \ \texttt{str} \quad \Gamma\vdash e_2\texttt{:}\ \ \texttt{str}}{\Gamma\vdash \texttt{\textasciicircum(}e_1\texttt{;}\ e_2\texttt{):}\ \ \texttt{str}}\ \texttt{T.5}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let(}x\texttt{;}\ e_1\texttt{;}\ e_2\texttt{):}\ \ \tau_2}\ \texttt{T.6}$$

Figure 3: Static Semantics of *MiniLang*

$$\frac{\dfrac{}{\vdash \texttt{num[24]:}\ \ \texttt{num}}\ \texttt{T.1} \qquad \dfrac{\dfrac{}{\texttt{hours :}\ \ \texttt{num} \vdash \texttt{hours:}\ \ \texttt{num}}\ \texttt{T.3} \quad \dfrac{}{\texttt{hours :}\ \ \texttt{num} \vdash \texttt{num[3]:}\ \ \texttt{num}}\ \texttt{T.1}}{\texttt{hours :}\ \ \texttt{num} \vdash \texttt{+(hours; num[3]):}\ \ \texttt{num}}\ \texttt{T.4}}{\vdash \texttt{let(hours; num[24]; +(hours; num[3])):}\ \ \texttt{num}}\ \texttt{T.6}$$

Figure 4: Example Typing Derivation

4 Dynamic Semantics

The dynamic semantics of *MiniLang* define a transition system for evaluating *MiniLang* expressions. In order to know when we are done evaluating an expression to a *value*, we need to define values. Values in *MiniLang* are either a single number or a single string.

$$\frac{}{\texttt{num}[n]\ \texttt{value}} \qquad \frac{}{\texttt{str}[s]\ \texttt{value}}$$

The inductive definition of the transition relation for evaluation expressions is shown in Figure 6. Rules `D.1-3` define how to evaluate additions. Rule `D.3` says that the addition of two single numbers *steps to* (evaluation step) a number that is the sum of those two numbers. Rule `D.1` says if an evaluation step can be performed on the left summand, then the addition expression steps to an expression that is the same except with the step performed on the left summand. Once we are done evaluating the left subexpression to a number, rule `D.2` allows us to evaluate the right subexpression. Rules `D.4-6` are analogous to rules `D.1-3` for string concatenation. Rule `D.7` says we evaluate the

$$\frac{\dfrac{}{\vdash \texttt{num[24]:}\ \ \texttt{num}}\ \texttt{T.1} \qquad \dfrac{\dfrac{}{\texttt{hours :}\ \ \texttt{num} \vdash \texttt{hours:}\ \ \texttt{num}}\ \texttt{T.3} \quad \dfrac{}{\texttt{hours :}\ \ \texttt{num} \vdash \texttt{str[abc]:}\ \ \texttt{str}}\ \texttt{T.1}}{\texttt{hours :}\ \ \texttt{num} \vdash \texttt{+(hours; str[abc]):}\ \ \mathit{Fail}}}{\vdash \texttt{let(hours; num[24]; +(hours; str[abc])}}$$

Figure 5: Example Type Failure

expression (e_1) that will be bound to variable of the `let` expression. Once e_1 becomes a value, then rule `D.8` says to we can evaluate the body (e_2) of the `let` expression by replacing the variable with that value.

$$\frac{e_1 \mapsto e_1'}{\texttt{+(}e_1\texttt{; }e_2\texttt{)} \mapsto \texttt{+(}e_1'\texttt{; }e_2\texttt{)}}\ \texttt{D.1} \qquad \frac{e_2 \mapsto e_2'}{\texttt{+(num[}n_1\texttt{]; }e_2\texttt{)} \mapsto \texttt{+(num[}n_1\texttt{]; }e_2'\texttt{)}}\ \texttt{D.2}$$

$$\frac{}{\texttt{+(num[}n_1\texttt{]; num[}n_2\texttt{])} \mapsto \texttt{num[}n_1 + n_2\texttt{]}}\ \texttt{D.3}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{\textasciicircum(}e_1\texttt{; }e_2\texttt{)} \mapsto \texttt{\textasciicircum(}e_1'\texttt{; }e_2\texttt{)}}\ \texttt{D.4} \qquad \frac{e_2 \mapsto e_2'}{\texttt{\textasciicircum(str[}s_1\texttt{]; }e_2\texttt{)} \mapsto \texttt{\textasciicircum(str[}s_1\texttt{]; }e_2'\texttt{)}}\ \texttt{D.5}$$

$$\frac{}{\texttt{\textasciicircum(str[}s_1\texttt{]; str[}s_2\texttt{])} \mapsto \texttt{str[}s_1\texttt{\textasciicircum}s_2\texttt{]}}\ \texttt{D.6}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{let(}x\texttt{; }e_1\texttt{; }e_2\texttt{)} \mapsto \texttt{let(}x\texttt{; }e_1'\texttt{; }e_2\texttt{)}}\ \texttt{D.7} \qquad \frac{e_1\ \texttt{value}}{\texttt{let(}x\texttt{; }e_1\texttt{; }e_2\texttt{)} \mapsto [e_1/x]e_2}\ \texttt{D.8}$$

Figure 6: Dynamic Semantics of *MiniLang*

5 Type Preservation

Type preservation or simply preservation is an important programming language property for eliminating certain errors that result in programs with undefined behavior. It establishes a key relationship between compile-time analysis (static semantics) and runtime behavior (dynamic semantics) of programs. The Preservation Theorem states that evaluating an expression does not change its type:

Theorem 1. ***(Preservation)*** *If $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.*

Preservation is important for real programming languages. Suppose, for example, that Java did not preserve types during evaluation, and consider what could go wrong in the following code segment:

```
int x;    // 4 bytes in Java
double y; // 8 bytes in Java
```

$$\texttt{x = }\qquad \underbrace{\texttt{x + 8}}_{\text{What if this evaluated to a double?}}$$

Different formats are used by compilers to represent values of different types. In order to interpret what is being represented by bytes at certain memory locations, the compiler has to know what type of value is at this location. Once the compiler knows what type of value is at a location, it knows the procedure to use to interpret the bytes at that location and the appropriate instructions to generate. However, if the type of the value at that location changed without the compiler knowing, then when the program tries to use the value at that location, what the program will do next is undefined. Hence, type preservation is needed to ensure that the behavior of the program throughout its execution is well-defined.

The remainder of this section proves preservation for *MiniLang*. We prove preservation by induction by showing that for each possible combination of the typing and evaluation judgments, preservation holds. The first proof case, for example, proves the preservation theorem for the case when typing rule `T.4` was applied to derive judgment $e : \tau$ and evaluation rule `D.3` was applied to derive judgment $e \mapsto e'$. The conclusion to prove for each proof case is judgment $e' : \tau$.

5.1 Base Case: (`T.4`, `D.3`) – Addition Case 1

$$\frac{\texttt{num}[n_1]\texttt{:}\ \ \texttt{num} \quad \texttt{num}[n_2]\texttt{:}\ \ \texttt{num}}{\texttt{+(num}[n_1]\texttt{;}\ \texttt{num}[n_2]\texttt{):}\ \ \texttt{num}}\ \texttt{T.4}$$

$$\frac{}{\texttt{+(num}[n_1]\texttt{;}\ \texttt{num}[n_2]\texttt{)}\ \mapsto\ \texttt{num}[n_1 + n_2]}\ \texttt{D.3}$$

Using rule `T.1`:

$$\frac{}{\texttt{num}[n_1 + n_2]\texttt{:}\ \ \texttt{num}}\ \texttt{T.1} \qquad \square$$

5.2 Inductive Case: (`T.4`, `D.1`) – Addition Case 2

$$\frac{e_1\texttt{:}\ \ \texttt{num} \quad e_2\texttt{:}\ \ \texttt{num}}{\texttt{+(}e_1\texttt{;}\ e_2\texttt{):}\ \ \texttt{num}}\ \texttt{T.4} \qquad \frac{e_1 \mapsto e_1'}{\texttt{+(}e_1\texttt{;}\ e_2\texttt{)}\ \mapsto\ \texttt{+(}e_1'\texttt{;}\ e_2\texttt{)}}\ \texttt{D.1}$$

We assume preservation holds for subexpressions. Hence, by the inductive hypothesis, $e_1\texttt{:}\ \ \texttt{num}$ and $e_1 \mapsto e_1'$ implies $e_1'\texttt{:}\ \ \texttt{num}$. Rule `T.4` gives us:

$$\frac{e_1'\texttt{:}\ \ \texttt{num} \quad e_2\texttt{:}\ \ \texttt{num}}{\texttt{+(}e_1'\texttt{;}\ e_2\texttt{):}\ \ \texttt{num}}\ \texttt{T.4} \qquad \square$$

5.3 Inductive Case: (`T.4`, `D.2`) – Addition Case 3

$$\frac{\texttt{num}[n_1]\texttt{:}\ \ \texttt{num} \quad e_2\texttt{:}\ \ \texttt{num}}{\texttt{+(num}[n_1]\texttt{;}\ e_2\texttt{):}\ \ \texttt{num}}\ \texttt{T.4} \qquad \frac{e_2 \mapsto e_2'}{\texttt{+(num}[n_1]\texttt{;}\ e_2\texttt{)}\ \mapsto\ \texttt{+(num}[n_1]\texttt{;}\ e_2'\texttt{)}}\ \texttt{D.2}$$

Since $e_2\texttt{:}\ \ \texttt{num}$ and $e_2 \mapsto e_2'$, by the inductive hypothesis, $e_2'\texttt{:}\ \ \texttt{num}$.
Rule `T.4` gives us:

$$\frac{\dfrac{}{\texttt{num}[n_1]\texttt{:}\ \ \texttt{num}}\ \texttt{T.1} \quad e_2'\texttt{:}\ \ \texttt{num}}{\texttt{+(num}[n_1]\texttt{;}\ e_2'\texttt{):}\ \ \texttt{num}}\ \texttt{T.4} \qquad \square$$

5.4 Concatenation Cases

The proof of the concatenation cases are analogous to the proofs of the addition cases by just replacing the `num`'s with `str`'s and the `+`'s with `^`'s. □

5.5 Substitution Lemma

Before we can prove the next case for our preservation proof, we need the Substitution Lemma. Informally, this lemma states that we can substitute subexpressions that are of the same type in an expression e without changing the type of e.

Lemma 1. ***(Substitution)*** *If* $y' : \tau$ *and* $y : \tau \vdash e : \tau'$*, then* $[y'/y]e : \tau'$*.*

This lemma can be proved by a typical proof by induction on the structure of e, so we skip this proof for brevity. Now we return back to the proof of preservation.

5.6 Base Case: (`T.6`, `D.8`) – Let Case 1

$$\frac{e_1 : \tau_1 \quad x : \tau_1 \vdash e_2 : \tau_2}{\texttt{let(}x;e_1;e_2\texttt{):}\quad \tau_2}\ \texttt{T.6}$$

$$\frac{e_1\ \texttt{value}}{\texttt{let(}x\texttt{;}\ e_1\texttt{;}\ e_2\texttt{)}\ \mapsto\ [e_1/x]e_2}\ \texttt{D.8}$$

Since $e_1 : \tau_1$ and $x : \tau_1 \vdash e_2 : \tau_2$, by substitution lemma, we have $[e_1/x]e_2 : \tau_2$. □

5.7 Inductive Case: (`T.6`, `D.7`) – Let Case 2

$$\frac{e_1 : \tau_1 \quad x : \tau_1 \vdash e_2 : \tau_2}{\texttt{let(}x;e_1;e_2\texttt{):}\quad \tau_2}\ \texttt{T.6}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{let(}x\texttt{;}\ e_1\texttt{;}\ e_2\texttt{)}\ \mapsto\ \texttt{let(}x\texttt{;}\ e_1'\texttt{;}\ e_2\texttt{)}}\ \texttt{D.7}$$

Since $e_1\texttt{:}\quad \tau_1$ and $e_1 \mapsto e_1'$, by the inductive hypothesis, $e_1'\texttt{:}\quad \tau_1$. Using rule T.6:

$$\frac{e_1' : \tau_1 \quad x : \tau_1 \vdash e_2 : \tau_2}{\texttt{let(}x;e_1';e_2\texttt{):}\quad \tau_2}\ \texttt{T.6} \qquad \square$$

We have completed the proof of preservation for *MiniLang*!

5.8 Final Remarks of Preservation Proof

We have proved preservation by showing that for each possible combination of the typing and evaluation judgments, the preservation theorem holds. How does one know when such a combination is possible? A combination is possible when there exists a unification of the *patterns* of the two judgments in question. For example, notice in the preservation proof, there was no case for typing rule `T.4` and evaluation rule `D.4`. The conclusion of rule `T.4` contains the expression `+(`e_1`;` e_2`)`.

We can think of the pattern of the expression $\texttt{+(}e_1\texttt{; } e_2\texttt{)}$ as that same expression except that the subexpressions or *parameters* e_1 and e_2 are variables that can be replaced with any other expression. In order for ($\texttt{T.4}$, $\texttt{D.4}$) to be a possible combination case for the preservation proof, there must exists a unifier for the pattern of $\texttt{+(}e_1\texttt{; } e_2\texttt{)}$ (conclusion expression of $\texttt{T.4}$) and the pattern of $\texttt{\^{}(}e_1\texttt{; } e_2\texttt{)}$ (expression to the left of $\mapsto$ in conclusion of $\texttt{D.4}$). Because the first symbols in each of those two expressions contain constant symbols ($\texttt{+}$ and $\texttt{\^{}}$) that differ, no such unification exists; so the ($\texttt{T.4}$, $\texttt{D.4}$) combination is not a possible case for the preservation proof. These remarks hint at the fact that these type of proofs can be automatically verified (e.g. by a proof assistant tool such as Twelf).

6 Progress Theorem

MiniLang expressions are evaluated to values by inputting them to the transition system defined by the dynamic semantics given in Figure 6. Each transition in the system *reduces* an expression or brings the expression closer to a value. However, not every irreducible expression is a value such as the following:

$$\texttt{+(num[5]; str[abc])} \mapsto \times$$

An expression e that is not a value, but for which there does *not* exists an e' such that $e \mapsto e'$ is said to be *stuck*. It should be the case that any stuck expression is ill-typed. Moreover, well-typed expressions do not get stuck. This property is expressed formally by the progress theorem.

Theorem 2. ***(Progress)*** *If $e : \tau$, then either e $\texttt{value}$ or there exists an expression e' such that $e \mapsto e'$.*

Progress is proved by induction on the typing rules. Again there a lot of cases, so for brevity we show just one case to get an idea of how to prove this theorem.

6.1 Inductive Case: ($\texttt{T.4}$) – Addition Case

$$\frac{e_1\texttt{:}\ \ \texttt{num} \quad e_2\texttt{:}\ \ \texttt{num}}{\texttt{+(}e_1\texttt{, } e_2\texttt{):}\ \ \texttt{num}}\ \texttt{T.4}$$

By the inductive hypothesis, since $e_1 : \texttt{num}$, either e_1 $\texttt{value}$ or there exists an expression e_1' s.t. $e_1 \mapsto e_1'$.

Suppose e_1 $\texttt{value}$. Then either $e_1 = \texttt{num[}n_1\texttt{]}$ or $e_1 = \texttt{str[}s_1\texttt{]}$. Since $e_1 : \texttt{num}$, then it must be the case that $e_1 = \texttt{num[}n_1\texttt{]}$.

Therefore, $e_1 = \texttt{num[}n_1\texttt{]}$ or there exists an expression e_1' such that $e_1 \mapsto e_1'$. Similarly, $e_2 = \texttt{num[}n_2\texttt{]}$ or $e_2 \mapsto e_2'$ for some e_2'.

Suppose $e_1 \mapsto e_1'$. Then:

$$\frac{e_1 \mapsto e_1'}{\texttt{+(}e_1\texttt{; } e_2\texttt{)} \ \mapsto\ \texttt{+(}e_1'\texttt{; } e_2\texttt{)}}\ \texttt{D.1}$$

Suppose $e_1 = \texttt{num}[n_1]$ and $e_2 \mapsto e_2'$. Then:

$$\frac{e_2 \mapsto e_2'}{\texttt{+(num}[n_1]\texttt{;}\ e_2\texttt{)} \ \mapsto\ \texttt{+(num}[n_1]\texttt{;}\ e_2'\texttt{)}}\ \texttt{D.2}$$

Suppose $e_1 = \texttt{num}[n_1]$ and $e_2 = \texttt{num}[n_2]$. Then:

$$\frac{}{\texttt{+(num}[n_1]\texttt{; num}[n_2]\texttt{)} \ \mapsto\ \texttt{num}[n_1 + n_2]}\ \texttt{D.3}$$

We have covered all of the nested cases for rule `T.4`. □

7 Twelf

In previous sections we showed how to formalize a programming language with a grammar for defining its syntax, static semantic (type checking) rules that defined which programs in a language are well-formed, and dynamic semantics defining how to execute programs in the language. A formal definition of a language enables the ability to prove properties about the language. We presented two important properties, preservation and progress, that relate the static semantics (compile-time analysis) with the dynamic semantics (runtime behavior).

We showed how to prove properties of languages by structural induction. These proofs are typically are long, tedious, and involve many cases. As a result, it is easy to make a mistake while writing such proofs. Furthermore, the length and detail of these proofs means it is easy for a human to miss mistakes while checking the proof. However, verifying and deriving proofs can (sometimes) be done automatically. Proof assistants such as Twelf [11], Coq [1], and Isabelle [9] are software tools that enable one to encode theory in their respective languages. Depending on the power of these tools, they can verify proofs of theorems or derive proofs of theorems. Using any of these tools requires a very large learning curve, and explaining all of the details of how they work is beyond the scope of this paper. To sketch an idea of how they work, the remainder of this section explains the Twelf encoding of *MiniLang*. The entire Twelf encoding is available online [15]. Further details on using Twelf can be found in other Twelf tutorials [12].

7.1 *MiniLang* Syntax in Twelf

File `syntax.elf` from [15] contains the Twelf encoding of *MiniLang*'s syntax. In Twelf, there are three *levels* of objects. *Kinds* are at the highest level. *Types* are at the second level. *Terms* are at the lowest level. Each type is of a certain kind. Each term is of a certain type. For example, we could think of the type `Array[Int]`, which is an array of integers, to be of kind `Array`. A term of type `Array[Int]` could be `[1, 2, 3, 4]`. In Twelf, one defines their language by defining kinds, types, and terms. The kind `type` is a primitive kind defined in the Twelf language. More information on Twelf's type system can be found in [18].

Next, we describe the statements in file `syntax.elf` using the terminology presented above.

On line 4, "`exp : type`" states that `exp` is a type of kind `type`. Similarly, line 7 defines `typ` to be type of kind `type`, and line 10 defines `nat` to be a type of kind `type`. Line 11 defines the term `z` to be of type `nat`. Line 12 defines `s` to be a function term of the function type from `nat`'s to `nat`'s. Lines 11 and 12 encode (Peano) natural numbers {`z, s(z), s(s(z)), s(s(s(z))), ...`}. Lines 14–22 define strings to be a sequence of the `char`'s separated by commas. The functions `enat` and `estr` are just ways to say that natural numbers and strings are also expressions. For example, `z` has type `nat`, but `(enat z)` has type `exp`. [1]

Lines 29–32 define the category of expressions with function types. For instance, "`add : exp -> exp -> exp.`" says that `add` is a function that takes into two `exp`'s and returns an `exp`. This corresponds to the grammar rule

$$\textit{Expression}\ e\ \ ::=\ \texttt{+(}e_1\texttt{;}\ e_2\texttt{)}$$

stating that the addition of two expressions is also an expression. [2] Similarly, "`cat : exp -> exp -> exp.`" says that `cat` is a function that takes into two `exp`'s and returns an `exp`; `cat` corresponds to the string concatenation of two subexpressions. The last two lines of `syntax.elf` define two terms `num` and `string` that represent the only two types in *MiniLang*.

The number of arguments a function term defined in Twelf takes may not be clear to a reader who is not familiar with *currying* [2, 4]. For example, the `add` function term looks like a function that takes in a single `exp` term and returns another function of type `exp -> exp`. Functions in Twelf are applied to terms by currying, where there is no distinction between functions f and f' when passing them two arguments if $f(x)$ returns another function $g(y)$ and $f'(x, y) = \underbrace{f(x)}_{g}(y)$. Passing a function multiple arguments is transformed into a chain of function calls, where each function in the chain is applied to a single argument. So a function that returns another function can be thought of as a function that takes in multiple arguments. Conversely, if a function f takes in multiple arguments $x_1, x_2, \ldots, x_n$ (where $n \geq 2$), then $f(t)$ can be thought of as a function g that is the same as f except that occurrences of variable x_1 in the body of f are replaced with term t in the body of g.

7.1.1 Representing terms w/ variables in Twelf using Higher-Order Abstract Syntax

The encoding of the `let` term in Twelf uses the technique of *higher-order abstract syntax* (HOAS) [6]. HOAS is a technique for representing abstract syntax trees with bound variables. The abstract syntax from Section 2 used to describe the grammar of *MiniLang* is actually *first-order abstract syntax* (FOAS). In FOAS, each AST has the form $o(t_1, t_2, \ldots, t_n)$, where o is an operator and $t_1, t_2, \ldots, t_n$ are each AST themselves. The operands of an operator correspond to subexpressions.

[1] It would be nice if Twelf had the notion of subtyping. Then we could just tell Twelf `nat` <: `exp`, so Twelf would infer that `z` should also have type `exp` without having to wrap it as `(enat z)`.

[2] There are a few minor things missing in the definition of *MiniLang* in this paper and its encoding in Twelf. For example, line 32 defines an expression for representing the length of a string (`len(`e`)`). Some details were left out for brevity.

For example, in "`+(`e_1`;` e_2`)`", e_1 and e_2 are operands/subexpressions of the `+` operator. The Twelf encoding of "`+(`e_1`;` e_2`)`" is "`add : exp -> exp -> exp.`", where the `add` corresponds to `+`, the leftmost `exp` corresponds to e_1, and the middle `exp` corresponds to e_2.

In HOAS, ASTs declare the variables they bind. For example, consider the AST $o(t_1, t_2, \ldots, t_n)$. In HOAS, each t_i has the form $x_1, x_2, \ldots x_k.t$, where t is an AST, each x_j is a variable bound in t, and $k \geq 0$; if $k = 0$, then t_i does not introduce new variables in the body, t. One advantage of knowing each variable introduced by an AST is that we can rename variables without changing the meaning of the AST.

Consider the `let` expression in *MiniLang* and its representation in FOAS:

$$\texttt{let(}x\texttt{; }e_1\texttt{; }e_2\texttt{)}$$

Recall that a `let` expression is evaluated by binding the variable x to the value of e_1 in e_2. Hence, a `let` expression can be modeled with the following HOAS:

$$\texttt{let(}e_1\texttt{; }x.e_2\texttt{)}$$

The "$x.e_2$" captures that variable x is bound in expression e_2. HOAS lets us know where variables are being bound. This lets us easily determine that the following two expressions are equivalent, since they only differ in variable names:

$$\texttt{let(3; }x\texttt{.+(}x\texttt{; 4))} \equiv \texttt{let(3; }y\texttt{.+(}y\texttt{; 4))}$$

Our Twelf encoding of the `let` expression is on line 36 of file `syntax.elf`:

```
let : exp -> (exp -> exp) -> exp.
```

This line encodes the HOAS version of `let`, which can be seen with the type of its second argument: `(exp -> exp)`. Usually, each operator such as `add` and `cat` only took in `exp` arguments, where the arguments represented subexpressions. However, the second argument to `let` is not just an ordinary subexpression but a subexpression that introduces a new variable. The second argument to `let` represents a higher-order AST of the form "$x.e_2$".

An AST that introduces new variables is represented in Twelf by a function. Functions in programming languages are really just terms with *holes*. The holes are represented by free variables, and these holes are filled in when these (terms w/ holes)/functions are applied to other terms. Hence, "$x.e_2$" can be thought of as the function or *lambda-abstraction* "$\lambda x.e_2$", where e_2 is the body of the function and x is a variable that is bound to a term in e_2. A term t of type `(exp -> exp)` will be a term of type `exp` that contains occurrences of a free variable x. Free variable x will be replaced with another term t' in the body of t when term/function t is applied to t'. Twelf's syntax for the function "$\lambda x.e$" is "`[`x`]` e". Figure 7 gives a concrete example of a `let` expression in concrete syntax and its corresponding representation in Twelf's HOAS.

Concrete Syntax	*Twelf HOAS*
$\texttt{let x = }\underbrace{\texttt{1 + 2}}_{e_1}\texttt{ in }\underbrace{\texttt{x + 3}}_{e_2}$	$\texttt{let }\underbrace{\texttt{(add 1 2)}}_{e_1}\ \underbrace{\texttt{([x] add x 3)}}_{x.e_2}$

Figure 7: Example Higher-Order Abstract Syntax in Twelf

7.2 *MiniLang* Static Semantics and Judgments in Twelf

File `typing.elf` from [15] contains the encoding of *MiniLang*'s static semantics in Twelf. Line 5, "`of : exp -> typ -> type.`", defines the relation `of` representing the judgment $e : \tau$. It defines `of` to be a function kind or *type family* [5, 13]. Type families are functions that return types instead of terms. The "`of`" relation can be thought of as (1) a function that returns types when applied to `exp` and `typ` terms or (2) as a set of types *indexed* by `exp` and `typ` terms.

Judgments are represented in Twelf as *dependent types* [10, 16, 3]. The `of` type family returns dependent types of kind `type`. Hence, function `of` returns judgments. Dependent types are types that include terms as one of its components. Example dependent types come from the set of n-dimensional vectors of real numbers denoted `Vec(`n`)`. For instance, suppose `3` is a term representing the number 3, $\mathbb{N}$ is a type representing the set of natural numbers, and `3` is of type $\mathbb{N}$. Also, assume `Vec` is a function that takes in a natural number n as input and returns a type representing the set of n-dimensional vectors. Then, `Vec(3)` is a dependent type representing 3-dimensional vectors of real numbers. A term of type `Vec(3)` is the vector $[7, 3, 4]$. Furthermore, function `Vec` denotes a type family indexed by natural numbers: `Vec(0)`, `Vec(1)`, `Vec(2)`, `Vec(3)`, ...

The `of` type family is indexed by `exp` and `typ` terms. The dependent type "`of` e τ" represents the judgment $e : \tau$. For example, dependent type "`of (enat z) num`" represents the proposition or judgment $z :$ `num`. A proof or derivation of a judgment/type is represented by a term of that type. An example proof will be presented later in this section.

Inference rules in Twelf are modeled as functions that return terms of dependent types. Consider the function term `of/nat` defined on line 7 of file `typing.elf`. Strings that begin with capital letters in Twelf are interpreted to be parameters. The `N` parameter in "`of (enat N) num`" is passed to the `enat` function term. Since `enat` only takes in values of type `nat`, so does `of/nat`. Twelf is able to infer that the `N` variable must be bound to a term of type `nat`.

At first, it seems the `of/nat` function returns types. For example, it seems "`of/nat z`" returns the dependent type "`of (enat z) num`". However, "`of/nat z`" actually returns a *term* of type "`of (enat z) num`". The term returned by "`of/nat z`" represents a derivation or proof of the judgment $z :$ `num`; this judgment is represented by type "`of (enat z) num`", which is the type of term "`of/nat z`". Hence, the `of/nat` term corresponds to rule `T.1`, which is the rule that allows us to derive that any natural number has type `num`. Similarly, the `of/str` function term takes in any string s and returns a derivation/term of a judgment/type stating that string s has type `string`.

Function term `of/add` models rule `T.4`. Premises of inference rules are represented by inputs that must be terms of a dependent type. For example, function `of/add` takes in two (explicit [8]) arguments of dependent types.[3] Variables `E1` and `E2` are bound to terms of type `exp`. Dependent type "`of E1 num`" represents the judgment e_1 : `num`. A term of type "`of E1 num`" represents a derivation of e_1 : `num`. Hence, `of/add` takes in a derivation of e_1 : `num` and a derivation of e_2 : `num` and returns a derivation/term of type "`of (add E1 E2) num`"; this type represents judgment `+(`e_1`;` e_2`) : num`.

7.2.1 Hypothetical Judgments in Twelf

Inference rules such as rule `T.6` involve the use of context-sensitive propositions or *hypothetical judgments* [7]. Hypothetical judgments are judgments that make use of hypothetical assumptions. For example, the second premise of rule `T.6`, "$\Gamma, x : \tau_1 \vdash e_2 : \tau_2$", is a hypothetical judgment. In order to prove that this proposition holds, judgment $e_2 : \tau_2$ needs to be proved; when proving $e_2 : \tau_2$, however, we can assume that the hypothetical assumption, "$\Gamma, x : \tau_1$" also holds. Specifically, judgment $e_2 : \tau_2$ needs to be proved in a context where we assume that variable x has type τ_1 and we assume that the other typing assumptions in Γ also hold.

Hypothetical judgments are modeled in Twelf as function types. Assumptions of a hypothetical judgment are the input types of a function. Encoding hypothetical judgments as function types is similar to the technique of higher-order abstract syntax discussed in Section 7.1.1, where syntactic terms with binders (e.g. the body of the `let` expression) are encoded as terms with holes or equivalently functions that take in other terms as inputs and return terms as outputs. In the case of hypothetical judgments, assumptions are the holes in the terms/proofs of those judgments. A proof of hypothetical judgments takes in proofs of hypothetical assumptions as inputs and returns a proof as output. Taking in hypothetical assumptions as inputs simulates extending the context with additional assumptions. The assumptions/inputs can be used in the body of a function to derive the desired output/conclusion.

Hypothetical judgments are encoded with a generalization of a function type called a *pi-type*, denoted $\Pi x : S.T$. Terms of pi-types are called *pi-abstractions.* First, a *lambda-abstraction*, denoted "$\lambda x : S.e$", is a function term of a more conventional kind of function type, $S \to T$. A function of this type takes in a term of type S and returns a term of type T. However, a pi-abstraction of pi-type $\Pi x : S.T$ would map a term s of type S to a term of type $[s/x]T$. That is, the return type of a pi-type can vary according to the argument supplied. Hence, if x is (syntactically) a part of return type T in the pi-type $\Pi x : S.T$, then a term returned by a function of that pi-type is a term of a dependent type. In that case, return type T depends on the input term x. If x is not a part of type T in $\Pi x : S.T$, then we abbreviate $\Pi x : S.T$ as $S \to T$. This signals that a lambda-abstraction can be of this type, since the return type does not involve the argument. Lastly, the pi-type $\Pi x : S.T$ is represented in Twelf syntax as `{x:S} T`.

[3] Function `of/add` actually takes in four arguments. The first two arguments are the `exp` terms `E1` and `E2`. However, these arguments are *implicit* arguments and typically do not need to be mentioned because they can be inferred from the last two *explicit* arguments of dependent types. For example, since the dependent type "`of E1 num`" includes `E1` as one if its components, Twelf can extract `E1` from this type. For more information on implicit and explicit arguments see [8].

A common Twelf coding convention is used in the return type of the `of/let` function term given on line 20 of file `typing.elf`:

```
of (let E1 ([x] E2 x)) T2
```

This expression could have been replaced with the shorter expression: "`of (let E1 E2) T2`". Twelf is able to infer that the type of variable `E2` in the shorter expression is function type "`exp -> exp`" (or equivalently, an `exp` with free variable occurrences). However, we applied a Twelf coding convention when we replaced `E2` with the equivalent function term `([x] E2 x)`. The wrapper function or *eta-expansion* of `E2`, `([x] E2 x)`, is used just to make it easier to see that `E2` is a function term that takes in a single argument.

The `of/let` term represents rule `T.6` and is of a *higher-order*[4] function type:

```
of/let :  ({x: exp} of x T1 -> of (E2 x) T2) ->
          of E1 T1 ->
          of (let E1 ([x] E2 x)) T2
```

The second argument type "`of E1 T1`" corresponds to the premise "$e_1 : \tau_1$". The type of `of/let`'s first argument is the pi-type: "`{x: exp} of x T1 -> of (E2 x) T2`"; let P denote this pi-type. A pi-abstraction of type P takes in two arguments: The first is an `exp` term, which will be bound to variable `x`. The second argument is a term of type "`of x T1`" representing a proof that `x` has type `T1`. Given two such terms, a pi-abstraction of type P should return a derivation of "`of (E2 x) T2`", where `(E2 x)` is the `E2` term with its hole filled in by the `exp` term that is bound to `x`.

The pi-type P represents the second premise "$\Gamma, x : \tau_1 \vdash e_2 : \tau_2$" of rule `T.6`. Passing a derivation `dx` of type "`of x T1`" to a pi-abstraction simulates extending the typing context with the assumption "$x : \tau_1$". The derivation `dx` of type "`of x T1`" can be used in the body of the pi-abstraction to return a term/proof of "`of (E2 x) T2`".

7.3 Twelf Theorems, Proofs, and Wrapup

Explaining all of the details of the Twelf code is beyond the scope of this paper. So in this section, we conclude the Twelf discussion with a brief overview of the remaining files of the Twelf *MiniLang* encoding, how theorems are specified in Twelf, presenting example Twelf proofs, and explaining how Twelf helps with proving theory about languages.

7.3.1 Twelf Project Files

Twelf projects are stored in directories containing a configuration file typically named `sources.cfg`. The `sources.cfg` file tells Twelf the files to read and the order in which to process them. The

[4] A higher-order function type is a function type where an input type is also a function type. Similarly, A higher-order function term such as `of/let` is a function that takes in a function as input.

evaluation.elf file contains the dynamic semantics of *MiniLang*. File preservation.elf contains the preservation theorem and its proof. File progress.elf contains the progress theorem and its proof. The other files are explained throughout Section 7.

7.3.2 Theorems are Function Types. Proofs of Theorems are Function Terms.

Theorems in Twelf are encoded in a similar manner to (for all)-(there exists) queries. For example, below is the encoding of the progress theorem.

```
%theorem
progress :
  forall* {E} {T}
  forall {O:of E T} exists {NS:not_stuck E}
  true.
```

The forall* line declares the variables in the theorem. The next line states that for every derivation of the typing judgment, of E T, where the derivation of that judgment is bound to O, there exists a derivation of the judgment that expression E is not stuck (NS:not_stuck E). The following two rules define the not_stuck judgment.[5]

not_stuck/val: not_stuck E <- value E. states that if E is value, then E is not stuck.

not_stuck/step: not_stuck E <- step E E'. states that if there exists an E' such that step E E', then E is not stuck as well.

Note that the above progress theorem is nothing more than a function type. Given a term/proof of the judgment of E T, a function of this type should return a term/proof of the judgment not_stuck E. Hence, a proof of the progress theorem/(function type) is a (total) function or term of the function type representing the theorem.

7.3.3 Checking and Deriving Proofs

The benefit of using Twelf is that it can check your proofs and occasionally derive proofs. File test_typing.elf, for example, provides two example judgments that can be derived automatically by Twelf.

```
%query * 1 D1 : of (estr (a , b , c , a , eps)) string.
%query * 1 D2 : of (add (enat (s z)) (enat z)) num.
```

The first query ask Twelf to derive that the type of the str[abca] is str, and save that derivation in D1. The second query ask Twelf to derive that the type of +(num[1]; num[0]) is num, and save that derivation in D2.

Below is the Twelf output from those queries.

[5] In Twelf, function types can be written using either arrows pointing to the left or pointing to the right.

```
loadFile test_typing.elf
[Opening file test_typing.elf]
%query * 1

of (estr (, a (, b (, c (, a eps))))) string.
---------- Solution 1 ----------
Empty Substitution.
D1 = of/str (, a (, b (, c (, a eps)))).
--------------------------------------------

%query * 1

of (add (enat (s z)) (enat z)) num.
---------- Solution 1 ----------
Empty Substitution.
D2 = of/add (enat z) (enat (s z)) (of/nat z) (of/nat (s z)).
```

Twelf finds derivations of both queries. For the first query, it finds the derivation

```
D1 = of/str (, a (, b (, c (, a eps)))).
```

It says that applying function `of/str` to term `str (, a (, b (, c (, a eps))))` returns a term/derivation of type/judgment “`of (estr (a , b , c , a , eps)) string`”.

For the second query, it finds the following derivation:

```
D2 = of/add (enat z) (enat (s z)) (of/nat z) (of/nat (s z)).
```

Derivation `D2` is described as follow:

1. Apply function `of/nat` to term `z` to return a term/derivation of “`of (enat z) num`”.
2. Apply function `of/nat` to term “`enat (s z)`” to return a term/derivation of “`of (enat (s z)) num`”.
3. Apply `of/add` to the derivations of “`of (enat z) num`” and “`of (enat (s z)) num`” to return a derivation of “`of (add (enat (s z)) (enat z)) num`”. The first argument `(enat z)` establishes that the third argument should be a term of type “`of (enat z) num`”; hence, the type of the third argument of `of/add` depends on the first argument passed to `of/add`. The situation is analogous for the second and fourth arguments of function `of/add`.

Twelf could not derive the proof of the preservation theorem, but it could let you know if your proof was incorrect. For example, lines 23–27 in `preservation.elf` prove preservation for the typing-evaluation rule combination (T.4, D.1). If those lines were removed, Twelf would return the following error output:

```
preservation.elf:69.8-69.11 Error:
Coverage error --- missing cases:
{E1:exp} {E2:exp} {E3:exp} {O1:of (add E1 E2) num} {S1:step E1 E3}
   {O2:of (add E3 E2) num}
   |- preservation O1 (step/add1 S1) O2.
```

The error message lets us know that we forgot to prove preservation for the case when we could derive "`of (add E1 E2) num`" and "`step (add E1 E2) (add E3 E2))`" ($\texttt{+(}e_1\texttt{;}\ e_2\texttt{)} \mapsto \texttt{+(}e_3\texttt{;}\ e_2\texttt{)}$). In this case, judgment "`step (add E1 E2) (add E3 E2))`" was derived by applying function (inference rule) `step/add1` to a derivation (`S1`) of type "`step E1 E3`". To prove preservation for this case, we would need to produce a derivation of type "`of (add E3 E2) num`". This would show that performing an evaluation step on the left summand of a well-typed `add` expression would not change the type of the resulting `add` expression.

In summary, Twelf can aid with deriving simple judgments and checking that proofs of language properties are correct. If a proof is not correct, Twelf will let us know that the proof contains a mistake and provide an error message to help "debug" the proof.

8 Summary

This paper gave an overview of type theory and how it applies to programming languages. It explained how important aspects of programming languages can be specified precisely and formally. It showed how properties of programming languages can be proven. It described how computers can aid in formal verification of properties. Lastly, it highlighted topics that the reader can investigate further to better understand foundations underlying programming languages.

References

[1] Coq. `http://coq.inria.fr/`.

[2] Currying. `http://www.haskell.org/haskellwiki/Currying`.

[3] Dependent types – Twelf. `http://twelf.plparty.org/wiki/Dependent_type`.

[4] Function Currying in Scala. `http://www.codecommit.com/blog/scala/function-currying-in-scala`.

[5] GHC/Type families. `http://www.haskell.org/haskellwiki/GHC/Type_families`.

[6] Higher-order abstract syntax. `http://twelf.plparty.org/wiki/Higher-order_abstract_syntax`.

[7] Hypothetical judgment. `http://twelf.org/wiki/Hypothetical_judgment`.

[8] Implicit and explicit parameters. http://twelf.plparty.org/wiki/Implicit_and_explicit_parameters.

[9] Isabelle. http://isabelle.in.tum.de/.

[10] Judgments as types. http://twelf.plparty.org/wiki/Judgments_as_types.

[11] Twelf. http://twelf.plparty.org/wiki/Main_Page.

[12] Twelf Tutorials. http://twelf.plparty.org/wiki/Tutorials.

[13] Type family. http://twelf.plparty.org/wiki/Type_family.

[14] Type system. http://en.wikipedia.org/wiki/Type_system.

[15] John Altidor. Twelf Tutorial and Twelf Encoding of Minilang. https://github.com/jgaltidor/twelf_tutorial.

[16] David Aspinall, Martin Hofmann, and Benjamin C. Pierce. *Chapter 2 of Advanced Topics in Types and Programming Languages.* The MIT Press, 2004. http://www.cis.upenn.edu/~bcpierce/attapl/main.html.

[17] Robert Harper. 15-814 Types and Programming Languages. http://www.cs.cmu.edu/~rwh/courses/typesys/.

[18] Robert Harper and Daniel Licata. Mechanizing metatheory in a logical framework. *J. Funct. Program.*, 17(4-5):613–673, July 2007.

[19] Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002. http://mitpress.mit.edu/catalog/item/default.asp?sid=AF8E3C7B-6915-4259-A53C-6D78276FF0AC&ttype=2&tid=8738.