

Type Theory Tutorial

John Altidor

Logic Seminar Lecture

- Brief Motivation for Type Systems.
- Example Type System/Programming Language (PL).
 - ▶ Presenting MiniLang: A simple programming language of numbers and strings.
 - ▶ Syntax
 - ▶ Static Semantics (Type Checking)
 - ▶ Dynamic (Operational) Semantics (Evaluation)
 - ▶ Safety Theorems: Preservation + Progress
- Twelf Tutorial
 - ▶ Mechanization of MiniLang

Create Language vs. Create Library

Create Language vs. Create Library

- Library Pros:
 - ▶ Library allows using existing language *infrastructure*

Create Language vs. Create Library

- Library Pros:
 - ▶ Library allows using existing language *infrastructure*
 - ▶ Smaller learning curve
 - Don't need to learn new language constructs.

Create Language vs. Create Library

- Library Pros:
 - ▶ Library allows using existing language *infrastructure*
 - ▶ Smaller learning curve
 - Don't need to learn new language constructs.
- Library Cons:
 - ▶ Errors difficult to *detect and debug* w/o a compiler.
 - ▶ Programs can enter undefined states (e.g., segmentation fault from reading a nonexistent field).

Create Language vs. Create Library

■ Library Pros:

- ▶ Library allows using existing language **infrastructure**
- ▶ Smaller learning curve
 - Don't need to learn new language constructs.

■ Library Cons:

- ▶ Errors difficult to **detect and debug** w/o a compiler.
 - ▶ Programs can enter undefined states (e.g., segmentation fault from reading a nonexistent field).
- ▶ Requirements **not checked** in the language of the library.
 - ▶ Leaking confidential information to unauthorized users.

Type Systems

- Type System = ***Formally defined*** language (calculus) with ***types***.
- ***Types*** = Properties/classification over terms (syntax) of a language.

Type Systems

- Type System = ***Formally defined*** language (calculus) with ***types***.
- ***Types*** = Properties/classification over terms (syntax) of a language.
- Precisely defining what a language means
 - ▶ Which programs are allowed in a language?
 - ▶ How does a program execute?
 - ▶ ...

Type Systems

- Type System = ***Formally defined*** language (calculus) with ***types***.
- ***Types*** = Properties/classification over terms (syntax) of a language.
- Precisely defining what a language means
 - ▶ Which programs are allowed in a language?
 - ▶ How does a program execute?
 - ▶ ...
- Enables ***proving properties*** about a language.
 - ▶ Program is always in a well-defined state throughout execution (no segmentation fault).
 - ▶ Can prove properties related to software requirements (e.g., information flow).

Type Systems

- Type System = **Formally defined** language (calculus) with **types**.
- **Types** = Properties/classification over terms (syntax) of a language.
- Precisely defining what a language means
 - ▶ Which programs are allowed in a language?
 - ▶ How does a program execute?
 - ▶ ...
- Enables **proving properties** about a language.
 - ▶ Program is always in a well-defined state throughout execution (no segmentation fault).
 - ▶ Can prove properties related to software requirements (e.g., information flow).
- Compiler **informs** programmers of errors at compile time.

Type Systems

- Type System = **Formally defined** language (calculus) with **types**.
- **Types** = Properties/classification over terms (syntax) of a language.
- Precisely defining what a language means
 - ▶ Which programs are allowed in a language?
 - ▶ How does a program execute?
 - ▶ ...
- Enables **proving properties** about a language.
 - ▶ Program is always in a well-defined state throughout execution (no segmentation fault).
 - ▶ Can prove properties related to software requirements (e.g., information flow).
- Compiler **informs** programmers of errors at compile time.
- Best explained with an example: **MiniLang**

- **Concrete syntax** is what humans write.
- **Abstract syntax** is what computers reason over.

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

Example Expressions

<i>Abstract</i>	<i>Concrete</i>
<code>+(num[5]; +(num[4]; num[3]))</code>	<code>5 + 4 + 3</code>
<code>^(str[john]; ^(x; str[doe]))</code>	<code>'john' ^ x ^ 'doe'</code>
<code>let(hours; num[24]; +(hours; num[33]))</code>	<code>let hours be 24 in hours+33</code>

Inference Rules

- Semantics of terms (ASTs) defined w/ inference rules.
- Rules have the following form.

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

- A rule with no premises is an axiom.

Static Semantics (Type Checking Rules)

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

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

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

Example Type Derivation

$$\frac{\frac{}{\vdash \text{num}[24] : \text{num}} \text{ T.1} \quad \frac{\frac{}{\text{hrs:num} \vdash \text{hrs} : \text{num}} \text{ T.3} \quad \frac{}{\text{hrs:num} \vdash \text{num}[3] : \text{num}} \text{ T.1}}{\text{hrs:num} \vdash +(hrs; \text{num}[3]) : \text{num}} \text{ T.4}}{\vdash \text{let}(hrs; \text{num}[24]; +(hrs; \text{num}[3])) : \text{num}} \text{ T.6}$$

Example Type Check Failure

$$\frac{\frac{}{\vdash \text{num}[24] : \text{num}} \text{ T.1} \quad \frac{\frac{}{\text{hrs:num} \vdash \text{hrs} : \text{num}} \text{ T.3} \quad \frac{}{\text{hrs:num} \vdash \text{str}[\text{abc}] : \text{str}} \text{ T.1}}{\text{hrs:num} \vdash +(\text{hrs}; \text{str}[\text{abc}]) : \text{Fail}}}{\vdash \text{let}(\text{hrs}; \text{num}[24]; +(\text{hrs}; \text{str}[\text{abc}]))}$$

Dynamic (Operational) Semantics (Evaluation)

- Defining how to “execute” expressions in MiniLang.
- Specifically, defining a **transition system**/relation $\mapsto$ between expressions to evaluate them to **values**.
- First, need to define values:

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

$$\frac{e_1 \mapsto e'_1}{+(e_1; e_2) \mapsto +(e'_1; e_2)} \text{ D.1}$$

$$\frac{e_2 \mapsto e'_2}{+(\text{num}[n_1]; e_2) \mapsto +(\text{num}[n_1]; e'_2)} \text{ D.2}$$

$$\frac{n_1 + n_2 = n_3}{+(\text{num}[n_1]; \text{num}[n_2]) \mapsto \text{num}[n_3]} \text{ D.3}$$

$$\frac{e_1 \mapsto e'_1}{\hat{\cdot}(e_1; e_2) \mapsto \hat{\cdot}(e'_1; e_2)} \text{ D.4}$$

$$\frac{e_2 \mapsto e'_2}{\hat{\cdot}(\text{str}[s_1]; e_2) \mapsto \hat{\cdot}(\text{str}[s_1]; e'_2)} \text{ D.5}$$

$$\frac{s_1 \hat{\cdot} s_2 = s_3}{\hat{\cdot}(\text{str}[s_1]; \text{str}[s_2]) \mapsto \text{str}[s_3]} \text{ D.6}$$

$$\frac{e_1 \mapsto e'_1}{\text{let}(x; e_1; e_2) \mapsto \text{let}(x; e'_1; e_2)} \text{ D.7}$$

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

Safety Theorem – Preservation

- If e is a well-typed expression that is not a value, then performing an evaluation step on e does not change its type.
- Formally, if $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.

Safety Theorem – Preservation

- If e is a well-typed expression that is not a value, then performing an evaluation step on e does not change its type.
- Formally, if $e : \tau$ and $e \mapsto e'$, then $e' : \tau$.
- Relates the compile-time analysis (type checking rules) with the run-time behavior (evaluation rules).
- Important property for real programming languages.

Preservation Motivation

What if Java did not preserve types during evaluation?

```
int x;    // 4 bytes in Java
```

```
double y; // 8 bytes in Java
```

```
x =       $\underbrace{x + 8}$ 
```

What if this evaluated to a double?

Preservation Proof – Addition Case 1

Proof by induction on the *possible* typing and evaluation combinations.

Case: (T.4, D.3)

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

$$\frac{n_1 + n_2 = n_3}{+ (\text{num}[n_1] ; \text{num}[n_2]) \mapsto \text{num}[n_3]} \text{ D.3}$$

Using rule T.1:

$$\frac{}{\text{num}[n_3] : \text{num}} \text{ T.1} \quad \square$$

Preservation Proof – Addition Case 2

Case: (T.4, D.1)

$$\frac{e_1 : \text{num} \quad e_2 : \text{num}}{+(e_1; e_2) : \text{num}} \text{ T.4} \qquad \frac{e_1 \mapsto e'_1}{+(e_1; e_2) \mapsto +(e'_1; e_2)} \text{ D.1}$$

We assume preservation holds for subexpressions. Hence, by the **inductive hypothesis**, $e_1 : \text{num}$ and $e_1 \mapsto e'_1$ imply $e'_1 : \text{num}$.

Rule T.4 gives us:

$$\frac{e'_1 : \text{num} \quad e_2 : \text{num}}{+(e'_1; e_2) : \text{num}} \text{ T.4} \quad \square$$

Preservation Proof – Addition Case 3

Case: (T.4, D.2)

$$\frac{\text{num}[n_1]: \text{num} \quad e_2: \text{num}}{+(\text{num}[n_1]; e_2): \text{num}} \text{ T.4}$$

$$\frac{e_2 \mapsto e'_2}{+(\text{num}[n_1]; e_2) \mapsto +(\text{num}[n_1]; e'_2)} \text{ D.2}$$

Since $e_2: \text{num}$ and $e_2 \mapsto e'_2$, by the inductive hypothesis, $e'_2: \text{num}$.

Rule T.4 gives us:

$$\frac{\overline{\text{num}[n_1]: \text{num}} \text{ T.1} \quad e'_2: \text{num}}{+(\text{num}[n_1]; e'_2): \text{num}} \text{ T.4} \quad \square$$

Preservation Proof – Remaining Cases

- Remaining cases in the preservation proof apply similar reasoning.
- We show one more case involving a common lemma.

Substitution Lemma

- For a case in the preservation proof, we need the ***Substitution Lemma***:
- In words, we can substitute subexpressions that are of the same type in an expression e without changing the type of e .

Substitution Lemma

- For a case in the preservation proof, we need the ***Substitution Lemma***:
- In words, we can substitute subexpressions that are of the same type in an expression e without changing the type of e .
- Formally:
If $\Gamma \vdash e' : \tau'$ and $\Gamma, y : \tau' \vdash e : \tau$, then $\Gamma \vdash [e'/y]e : \tau$.
- Proof of this lemma by induction on the structure of e .

Preservation Proof – Let Case

Case: (T.6, D.8)

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

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

Since $e_1 : \tau_1$ and $x : \tau_1 \vdash e_2 : \tau_2$, by the **Substitution Lemma**, we have $[e_1/x]e_2 : \tau_2$. $\square$

We have completed the proof of preservation!

- Preservation + Progress = Type Safety
 - ▶ Progress theorem and proof presented in the paper.

Type Safety

- Preservation + Progress = Type Safety
 - ▶ Progress theorem and proof presented in the paper.
- Type safety ensures program behavior is well-defined throughout its execution.

- Preservation + Progress = Type Safety
 - ▶ Progress theorem and proof presented in the paper.
- Type safety ensures program behavior is well-defined throughout its execution.
- Proving language properties is important (e.g., ruling out certain errors, publishing).

- Preservation + Progress = Type Safety
 - ▶ Progress theorem and proof presented in the paper.
- Type safety ensures program behavior is well-defined throughout its execution.
- Proving language properties is important (e.g., ruling out certain errors, publishing).
- But proofs are long, error-prone, and difficult to validate.
- Automated support for ***deriving proofs*** and ***checking proofs*** of language properties.
 - ▶ Twelf, Rocq (formerly Coq), Isabelle, Agda, ...

Summary

- Programming languages can be defined using formal mathematical specification.
 - ▶ Which programs are allowed.
 - ▶ How a program executes.
- Formal specification enables proving language properties.
- Type system = formally defined language with types.
- Type safety theorems (e.g., preservation) establish a relationship between compile-time analysis and run-time behavior.