Type Theory Tutorial

John Altidor

Logic Seminar Lecture

John Altidor Type Theory Tutorial 1/24

- 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*

John Altidor Type Theory Tutorial 3/24

@ ▶ < E ▶ < E ▶

Library allows using existing language infrastructure

★ E ► ★ E ►

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

▲ 문 ▶ | ▲ 문 ▶

- 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 non-existing field).

白 ト イヨト イヨト

- 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 non-existing field).
 - Requirements not checked in the language of the library.
 - Leaking confidential information to unauthorized users.

同 と く き と く き と

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

· < @ > < 글 > < 글 > · · 글

- 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?
 - ▶ ...

(本部) (本語) (本語) (語)

- 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).

イロン イ部ン イヨン イヨン 三日

- 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.

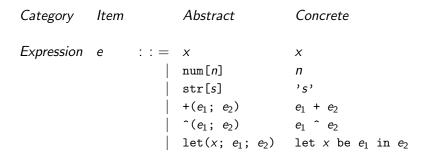
・ロン ・回 と ・ ヨ と ・ ヨ と

- 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

・ロン ・回 と ・ ヨ と ・ ヨ

4/24

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



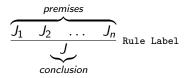
イロン イヨン イヨン イヨン

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

◆□ > ◆□ > ◆臣 > ◆臣 > ○

Semantics of terms (ASTs) defined w/ inference rules.

Rules have the following form.



• No premises means axiom.

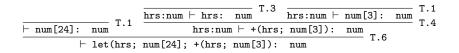
Static Semantics (Type Checking Rules)

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

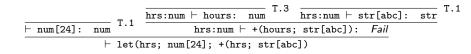
$$\frac{\Gamma \vdash e_1: \text{ num } \Gamma \vdash e_2: \text{ num }}{\Gamma \vdash +(e_1; e_2): \text{ num }} \text{ T.4 } \frac{\Gamma \vdash e_1: \text{ str } \Gamma \vdash e_2: \text{ str }}{\Gamma \vdash \hat{}(e_1; e_2): \text{ str }} \text{ T.5}$$

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

副 🕨 🖉 🖻 🖌 🖉 🖻 👘



(4回) (4回) (4回)



イロン イヨン イヨン イヨン

Dynamic (Operational) Semantics (Evaluation)

- Defining how to "execute" expressions in *MiniLang*.
- Specifically, defining a **transition system**/relation → between expressions to evaluate them to **values**.
- First, need to define values:

num[n] value str[s] value

伺 とう ヨン うちょう

Dynamic Semantics – Numerical Addition

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

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

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

John Altidor Type Theory Tutorial 12/24

□ ▶ 《 臣 ▶ 《 臣 ▶ …

Dynamic Semantics – String Concatenation

$$\frac{e_2 \mapsto e_2'}{(\operatorname{str}[s_1]; e_2) \mapsto (\operatorname{str}[s_1]; e_2')} \text{ D.5}$$

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

John Altidor Type Theory Tutorial 13/24

白 と く ヨ と く ヨ と …

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

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

John Altidor Type Theory Tutorial 14/24

副 🕨 🖉 🖻 🖌 🖉 🕨 👘

- 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 $e' : \tau$.

- 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 langauges.

What if Java did not preserve types during evaluation?

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

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

→ 注→ 注

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

Case: (T.4, D.3)

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

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

Using rule T.1:

$$\overline{\operatorname{num}[n_3]: \operatorname{num}^{T.1}}$$

 æ

→ Ξ → ...

Case: (T.4, D.1)

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

We assume preservation holds for subexpressions. Hence, by the **inductive hypothesis**, e_1 : num and $e_1 \mapsto e'_1$ implies e'_1 : num. Rule T.4 gives us:

$$\frac{e_1': \text{ num } e_2: \text{ num }}{+(e_1'; e_2): \text{ num }} \text{ T.4} \quad \Box$$

白 と く ヨ と く ヨ と …

Preservation Proof – Addition Case 3

Case: (T.4, D.2)

$$\frac{\operatorname{num}[n_{1}]: \operatorname{num} e_{2}: \operatorname{num}}{+(\operatorname{num}[n_{1}]; e_{2}): \operatorname{num}} T.4$$

$$\frac{e_{2} \mapsto e_{2}'}{+(\operatorname{num}[n_{1}]; e_{2}) \mapsto +(\operatorname{num}[n_{1}]; e_{2}')} D.2$$

Since e_2 : num and $e_2 \mapsto e_2'$, by the inductive hypothesis, e_2' : num.

Rule T.4 gives us:

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

글 > 글

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

- 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 $\Gamma, \gamma : \tau' \vdash e : \tau$, then $\Gamma \vdash [e'/\gamma]e : \tau$.
- Proof of this lemma by induction on the structure of e.

Case: (T.6, D.8)

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

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

Since $e_1 : \tau_1$ and $x : \tau_1 \vdash e_2 : \tau_2$, by **substitution lemma**, we have $[e_1/x]e_2 : \tau_2$. \Box We have completed the proof of preservation!

2

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

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

▲圖▶ ▲屋▶ ▲屋▶

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

▲□→ ▲ □→ ▲ □→

- Preservation + Progress = Type Safety
 - Progress theorem and proof presented in them paper.
- Type safety ensure program behavior is well-defined throughout its execution.
- Proving language properties are 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, Coq, Isabelle, Agda, ...

(本部) (本語) (本語) (語)

- 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 relationship between compile-time analysis and run-time behavior.

- 4 回 2 - 4 □ 2 - 4 □