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

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 non-existing field).
▶ Requirements not checked in the language of the library.
▶ Leaking confidential information to unauthorized users.
Library Pros:
- Library allows using existing language infrastructure

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

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

John Altidor
Type Theory Tutorial 4/24
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.

<table>
<thead>
<tr>
<th>Category</th>
<th>Item</th>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>$e$</td>
<td>:: $=$</td>
<td>$x$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\text{num}[n]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\text{str}[s]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$(e_1; e_2)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$(e_1; e_2)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>let($x; e_1; e_2$)</td>
</tr>
</tbody>
</table>
Example Expressions

<table>
<thead>
<tr>
<th>Abstract</th>
<th>Concrete</th>
</tr>
</thead>
<tbody>
<tr>
<td>+(num[5]; +(num[4]; num[3]));</td>
<td>5 + 4 + 3</td>
</tr>
<tr>
<td>^(str[john]; ^(x; str[doe]));</td>
<td>'john' ^ x ^ 'doe'</td>
</tr>
<tr>
<td>let(hours; num[24]; +(hours; num[3]));</td>
<td>let hours be 24 in hours+24</td>
</tr>
</tbody>
</table>
Inference Rules

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

\[
\begin{array}{c}
\text{premises} \\
J_1, J_2, \ldots, J_n \\
\hline \\
\text{conclusion}
\end{array}
\]

- No premises means axiom.
Static Semantics (Type Checking Rules)

\[ \Gamma \vdash \text{num}[n]: \text{num} \quad \Gamma \vdash \text{str}[s]: \text{str} \]

\[ (x, \tau) \in \Gamma \]

\[ \Gamma \vdash x : \tau \]

\[ \Gamma \vdash e_1: \text{num} \quad \Gamma \vdash e_2: \text{num} \]

\[ \Gamma \vdash + (e_1; e_2): \text{num} \]

\[ \Gamma \vdash e_1: \text{str} \quad \Gamma \vdash e_2: \text{str} \]

\[ \Gamma \vdash ^{(e_1; e_2)}: \text{str} \]

\[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2: \tau_2 \]

\[ \Gamma \vdash \text{let}(x; e_1; e_2): \tau_2 \]
Example Type Derivation

⊢ \text{let}(\text{hrs}; \text{num}[24]; +\text{hrs}; \text{num}[3]): \text{num} \quad \text{T.6}

\begin{align*}
\vdash \text{num}[24]: \text{num} & \quad \text{T.1} \\
\vdash \text{hrs}: \text{num} & \quad \text{T.3} \\
\vdash \text{hrs}: \text{num} \vdash \text{num}[3]: \text{num} & \quad \text{T.4} \\
\vdash \text{hrs}: \text{num} \vdash +\text{hrs}; \text{num}[3]: \text{num} & \quad \text{T.1} \quad \text{T.4} \\
\end{align*}
Example Type Check Failure

⊢ num[24]: num T.1
⊢ hours: num T.3
⊢ str[abc]: str T.1

\[\vdash \text{let}(\text{hrs}; \text{num}[24]; + (\text{hrs}; \text{str}[\text{abc}])): \text{Fail} \]

\[\vdash \text{hrs}: \text{num} \vdash \text{hours}: \text{num} \quad \vdash \text{str}[\text{abc}]: \text{str} \]

\[\vdash \text{num}[24]: \text{num} \quad \vdash \text{hrs}: \text{num} \vdash + (\text{hours}; \text{str}[\text{abc}]): \text{Fail} \]

John Altidor
Type Theory Tutorial 10/24
- Defining how to “execute” expressions in *MiniLang*.
- Specifically, defining a transition system/relation $\rightarrow$ between expressions to evaluate them to values.
- First, need to define values:
  
<table>
<thead>
<tr>
<th>num[$n$] value</th>
<th>str[$s$] value</th>
</tr>
</thead>
</table>

Dynamic Semantics – Numerical Addition

\[
\begin{align*}
\text{D.1} & \quad e_1 &\triangleright& e'_1 \\
& &\quad + (e_1; e_2) &\triangleright& + (e'_1; e_2) \\
\text{D.2} & \quad e_2 &\triangleright& e'_2 \\
& &\quad + (\text{num}[n_1]; e_2) &\triangleright& + (\text{num}[n_1]; e'_2) \\
\text{D.3} & \quad n_1 + n_2 &= n_3 \\
& &\quad + (\text{num}[n_1]; \text{num}[n_2]) &\triangleright& \text{num}[n_3]
\end{align*}
\]
Dynamic Semantics – String Concatenation

\[ e_1 \mapsto e'_1 \]
\[ \langle e_1; e_2 \rangle \mapsto \langle e'_1; e_2 \rangle \]  \hspace{1cm} D.4

\[ e_2 \mapsto e'_2 \]
\[ \langle \text{str}[s_1]; e_2 \rangle \mapsto \langle \text{str}[s_1]; e'_2 \rangle \]  \hspace{1cm} D.5

\[ s_1 \text{^} s_2 = s_3 \]
\[ \langle \text{str}[s_1]; \text{str}[s_2] \rangle \mapsto \text{str}[s_3] \]  \hspace{1cm} D.6
Dynamic Semantics – Let Expressions

\[
\begin{align*}
\frac{e_1 \mapsto e'_1}{\text{let}(x; e_1; e_2) \mapsto \text{let}(x; e'_1; e_2)} & \quad \text{D.7} \\
\frac{e_1 \text{ value}}{\text{let}(x; e_1; e_2) \mapsto [e_1/x]e_2} & \quad \text{D.8}
\end{align*}
\]
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.
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?

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

x = x + 8
```

What if this evaluated to a double?
Proof by induction on the possible typing and evaluation combinations.

Case: (T.4, D.3)

\[
\begin{align*}
\text{num}[n_1]: \text{num} & \quad \text{num}[n_2]: \text{num} \\
+ (\text{num}[n_1]; \text{num}[n_2]): \text{num} & \quad \text{T.4}
\end{align*}
\]

\[
\begin{align*}
n_1 + n_2 &= n_3 \\
+ (\text{num}[n_1]; \text{num}[n_2]) &\mapsto \text{num}[n_3] \quad \text{D.3}
\end{align*}
\]

Using rule T.1:

\[
\begin{align*}
\text{num}[n_3]: \text{num} & \quad \text{D.3}
\end{align*}
\]
Case: (T.4, D.1)

\[
\frac{e_1: \text{num} \quad e_2: \text{num}}{+(e_1; e_2): \text{num}} \quad \text{T.4}
\]

\[
\frac{e_1 \mapsto e_1'}{+(e_1; e_2) \mapsto +(e_1'; e_2)} \quad \text{D.1}
\]

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

\[
\frac{e_1': \text{num} \quad e_2: \text{num}}{+(e_1'; e_2): \text{num}} \quad \text{T.4}
\]
Preservation Proof – Addition Case 3

Case: (T.4, D.2)

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

\[
\frac{
e_2 \leftrightarrow e_2'
}{
+ (\text{num}[n_1]; e_2) \leftrightarrow + (\text{num}[n_1]; e_2')
}\quad \text{D.2}
\]

Since \(e_2: \text{num}\) and \(e_2 \leftrightarrow e_2'\), by the inductive hypothesis, \(e_2': \text{num}\).

Rule T.4 gives us:

\[
\frac{
\text{num}[n_1]: \text{num} \quad e_2': \text{num}
}{
+ (\text{num}[n_1]; e_2'): \text{num}
}\quad \text{T.1}
\]

\[
\frac{
}{
}\quad \text{T.4}
\]

□
Remaining cases in 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 $e$. 

Formally:

Given $\Gamma \vdash e' : \tau'$ and $\Gamma, y : \tau' \vdash e : \tau$, then $\Gamma \vdash [e' / y]e : \tau$. 

Proof of this lemma by induction on the structure of $e$. 

John Altidor

Type Theory Tutorial
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)

\[
\begin{align*}
\text{let}(x; e_1; e_2): & \quad \tau_2 \\
\text{let}(x; e_1; e_2) \mapsto & \quad [e_1/x]e_2
\end{align*}
\]

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

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

Type safety ensures 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, ...
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.