Type Theory Tutorial

John Altidor

Logic Seminar Lecture

Outline

- 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 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.

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**

MiniLang Syntax

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

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

Example Expressions

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[3])`	`let hours be 24 in hours+24`

Inference Rules

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

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

- No premises means axiom.

Static Semantics (Type Checking Rules)

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

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

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

Example Type Derivation

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

Example Type Check Failure

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

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:

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

Dynamic Semantics – Numerical Addition

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

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

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

Dynamic Semantics – String Concatenation

$$\frac{e_1 \mapsto e_1'}{\texttt{\^{}(}e_1\texttt{;}\ e_2\texttt{)} \mapsto \texttt{\^{}(}e_1'\texttt{;}\ e_2\texttt{)}}\ \texttt{D.4}$$

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

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

Dynamic Semantics – Let Expressions

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

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

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 langauges.

Preservation Motivation

What if Java did not preserve types during evaluation?

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

```
x =       x + 8
```

$\underbrace{\texttt{x + 8}}$ 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)

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

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

Using rule T.1:

$$\frac{}{\texttt{num}[n_3]: \ \texttt{num}}\ \texttt{T.1} \qquad \square$$

Preservation Proof – Addition Case 2

Case: (T.4, D.1)

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

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

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

Preservation Proof – Addition Case 3

Case: ($\texttt{T.4}$, $\texttt{D.2}$)

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

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

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

Rule $\texttt{T.4}$ gives us:

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

Preservation Proof – Remaining Cases

- 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.

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)

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

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

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

We have completed the proof of preservation!

Type Safety

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

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.
- Proving language properties are important (e.g. ruling out certain errors, publishing).

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.
- 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, ...

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