Type Theory
(the very basics)

Lecture #5

Spring, 2019

236608

Software Synthesis and Automated Reasoning
Reasoning

Type Theory (basics)

λ-calculus

Dependent Types

Programming by Example

Syntax Guided Synthesis

Counterexample Guided Inductive Synthesis

Type Directed Synthesis

Refinement Types

Axiomatic Semantics

Satisfiability Modulo Theory
What is Type Theory?

[1971]
Per Martin-Löf

Type Theory is a logic.

- but, being based on λ-calculus, is bares close similarity to programming languages.
Type Theory is a Logic

• As a logic, it is interested in making judgements of the form:
  ▶ $A$ is a set
  ▶ $A_1$ and $A_2$ are equal sets
  ▶ $a$ is in an element of set $A$
  ▶ $a_1$ and $a_2$ are equal elements in $A$
Judgement Forms

• To know that $A$ is a set is to know how to form the *canonical elements* in the set and under what conditions two canonical elements are equal.

**Set of natural numbers**

<table>
<thead>
<tr>
<th>Inductive</th>
<th>nat $\triangleleft$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 : nat</td>
<td></td>
</tr>
<tr>
<td>S : nat $\rightarrow$ nat</td>
<td></td>
</tr>
</tbody>
</table>

0
1 $\triangleleft$ S 0
2 $\triangleleft$ S (S 0)
3 $\triangleleft$ S (S (S 0))

**Set of lists**

<table>
<thead>
<tr>
<th>Inductive</th>
<th>List E $\triangleleft$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nil : List E</td>
<td></td>
</tr>
<tr>
<td>Cons : E $\rightarrow$ List E $\rightarrow$ List E</td>
<td></td>
</tr>
</tbody>
</table>

[] $\triangleleft$ Nil
[a$_1$] $\triangleleft$ Cons a$_1$ Nil
[a$_1$, a$_2$] $\triangleleft$ Cons a$_1$ (Cons a$_2$ Nil)
[a$_1$, a$_2$, a$_3$] $\triangleleft$ Cons a$_1$ (Cons a$_2$ (Cons a$_3$ Nil))

⋮
Judgement Forms

• If $A$ is a set then to know that $a \in A$ is to know that $a$, when evaluated, yields a canonical element of $A$ as value.

\[
\text{(fix } \lambda h. \text{ match } n \text{ with } \\
0 \Rightarrow \text{Nil} \\
S k \Rightarrow \text{Cons } n (h k) \text{ end) 8 : List nat}
\]

› We know this from our typing rules and from properties of the type system (progress & preservation)
Judgement Forms

• Quite unsurprisingly then,

  ‣ Sets are characterized by types,
  ‣ Membership is characterized by typing rules.
Propositions in Type Theory

• Propositions are **types**.
  - A member of the type represents a **proof** of the proposition.
    (sometimes called a **witness**.)

\[
\begin{align*}
  t & : 3 > 2 \\
  3 > 2 & : \mathbb{P}
\end{align*}
\]

  - Propositions have a special type called **Prop**, or \( \mathbb{P} \).
Propositional Logic

Logical \textit{True} \quad \top \quad \text{Inductive} \text{ True} \triangleq \text{I} : \text{True}

Logical \textit{False} \quad \bot \quad \text{Inductive} \text{ False} \triangleq \leftarrow \text{nothing} \rightarrow

\text{traditional logic} \quad \text{type theory}
Propositional Logic

Conjunction
\[ A \land B \]

Inductive
\[ \text{Pair } A \rightarrow B \rightarrow \text{Pair } A \land B \]
product types \((A \times B)\)

Disjunction
\[ A \lor B \]

Inductive
\[ \text{Either } A \rightarrow B \rightarrow \text{Either } A \lor B \]
sum types \((A + B)\)

traditional logic

type theory
Propositional Logic

Implication

\[ P \rightarrow Q \]

Negation

\[ \neg P \]

\[ P \rightarrow Q \]

function types!

\[ P \rightarrow \text{False} \]

function to vacuity

traditional logic
type theory
First-Order Logic

Function Symbols
\[ f : S_1 \times S_2 \rightarrow S_3 \]

Predicates
\[ R : S_1 \times S_2 \]

well... functions.

traditional logic

type theory

\[ f : S_1 \rightarrow S_2 \rightarrow S_3 \]

\[ R : S_1 \rightarrow S_2 \rightarrow \mathbb{P} \]

functions to Prop
Equality
\[ t_1 = t_2 \]

**Inductive**
\[
\begin{align*}
\text{eq} \, A : A \rightarrow A \rightarrow \mathbb{P} & \equiv \\
\text{eq_refl} : \forall a : A. \text{eq} \, a \, a
\end{align*}
\]
definitional equality

**eq [nat] 8 8 = 8 = 8 : \mathbb{P}**

**eq_refl 8 : 8 = 8**

I am a type!

eq_refl 8 is a proof that 8 = 8 is valid.

traditional logic

type theory
Quantifiers...

Universal Quantifier
\[ \forall x : S. \, P(x) \]

Existential Quantifier
\[ \exists x : S. \, P(x) \]

(dependent function types)

(they even have the same syntax...)

Inductive
\[ \text{Inductive } \text{Sig } S \, P \triangleq \]
\[ \text{sig} : \forall x : S. \, P(x) \rightarrow \text{Sig } S \, P \]

(dependent pair types)

\(\Pi\) types)

\(\Sigma\) types)

traditional logic  type theory
Quantifiers...

\[ P \rightarrow Q \]
function types

\[ \forall x : S. P x \]
dependent function types

\[ \land \]

**Inductive** Pair A B \( \triangleq \)

pair : A \( \rightarrow \) B \( \rightarrow \) Pair A B

product types (A * B)

\[ \exists \]

**Inductive** Sig S P \( \triangleq \)

sig : \( \forall x : S. P x \rightarrow Sig S P \)
dependent pair types

*type theory*
Quantifiers: an aside

• In Type Theory:

\( \forall \)  is a dependent generalization of \( \rightarrow \)

\( \exists \)  is a dependent generalization of \( \land \)
High-Order Quantifiers

Universal Quantifier

\[ \forall x : S. \ P(x) \]

dependent function types

\[ \forall A B \ (f : A \rightarrow B) \ x \ y. \ x = y \rightarrow f \ x = f \ y : \ P \]

dependent function types

traditional logic  type theory
Beyond First-Order Logic

**Induction?**

For \( n = 0 \): ...
Assume for \( n = k \),
prove for \( n = k+1 \):
...

**induction schema**

\[
\begin{align*}
P(0) & \quad P(k) \to P(k+1) \\
\hline&P(n)
\end{align*}
\]

**induction principle**

\[
\text{nat_ind} : \forall P : \mathbb{N} \to \mathbb{P}. \\
P(0) \to (\forall k : \mathbb{N}. P(k) \to P(S k)) \to \forall n : \mathbb{N}, P(n)
\]
Curry-Howard Correspondence

Haskell B. Curry [1969]

William A. Howard [1934, 1958]
## Curry-Howard Correspondence

<table>
<thead>
<tr>
<th>Logic</th>
<th>Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propositions / statements</td>
<td>Types</td>
</tr>
<tr>
<td></td>
<td>Programs</td>
</tr>
<tr>
<td>Proofs</td>
<td></td>
</tr>
<tr>
<td>True statement</td>
<td>Unit type</td>
</tr>
<tr>
<td>False statement</td>
<td>Bottom (empty) type</td>
</tr>
<tr>
<td>Conjunction</td>
<td>Product type (pair)</td>
</tr>
<tr>
<td>Disjunction</td>
<td>Sum type (either)</td>
</tr>
<tr>
<td>Implication</td>
<td>Function type</td>
</tr>
<tr>
<td>Modus ponens</td>
<td>Function application</td>
</tr>
<tr>
<td>Universal quantification</td>
<td>Dependent function type</td>
</tr>
<tr>
<td>Existential quantification</td>
<td>Dependent pair type</td>
</tr>
<tr>
<td>Induction</td>
<td>Recursion</td>
</tr>
</tbody>
</table>
Inductively Defined Predicates

**Inductive** \(eq\) : \(A \rightarrow A \rightarrow \mathbb{P} \triangleq\)
- \(eq\_refl : \forall a : A. \ eq\ a\ a\)

**Inductive** \(\le\) : \(\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{P} \triangleq\)
- \(\le\_n : \forall a : \mathbb{N}. \ le\ a\ a\)
- \(\le\_S : \forall a\ b : \mathbb{N}. \ le\ a\ b \rightarrow \ le\ a\ (S\ b)\)

\(a = b \triangleq eq\ a\ b\)

\(a \le b \triangleq le\ a\ b\)
3-Step Plan for Proving Propositions

- **Step 1.** Write program
  
  ```
  fix \lambda IH n. 
  match n with 
  0 \Rightarrow le_n 0 
  S k \Rightarrow le_S 0 k (IH k) 
  ```

- **Step 2.** Type-check
  
  : \forall n : \mathbb{N}. 0 \leq n : P

- **Step 3.** Profit!

: 0 \leq 0

: 0 \leq S k
3-Step Plan for Proving Propositions

```haskell
fix \(\lambda (\text{IH} : \forall n : \mathbb{N}. \ 0 \leq n) \ n.\)
  IH n
```

- **Step 1.** Write program
- **Step 2.** Type-check
- **Step 3.** Profit?

What about this fine program, though?

\[ \forall n : \mathbb{N}. \ 0 \leq n : \mathbb{P} \]
3-Step Plan for Proving Propositions

- **Step 1.** Write program
  - fix \( \lambda \text{IH} \ n. \)
    - match \( n \) with
      - \( 0 \) => \( \text{le}_n 0 \)
      - \( S \ k \) => \( \text{le}_S 0 \ k \) (IH \( k \))

- **Step 2.** Type-check:
  - \( : 0 \leq 0 \)
  - \( : 0 \leq S \ k \)
  - \( : \forall n : \mathbb{N}. \ 0 \leq n : \ P \)

- **Step 3.** Termination check
Constructivism

• Type Theory is a form of constructive mathematics.
  ‣ To prove $A \lor B$: one has to show how to determine whether $A$ is true or $B$ is true.
  ‣ To prove $\exists x. P$: one has to show how to find a value of $x$ that satisfies $P$.

Compute (algorithm!)
Implementation

Coq

Thierry Coquand

CoC

[1988]

Calculus of Constructions

later CIC

Calculus of Inductive Constructions

[1989]
Implementation

Coq

Type

Set

Prop

nat

list A

le x y

A \wedge B
Lab #4

• Prove some properties of natural numbers
  ‣ Follow the lemmas in gcd.v and fill in the proofs

• Prove (partial) correctness of a “softened” version of Euclid’s GCD algorithm.
  ‣ (gcd_correct) If gcd terminates, then the return value is a GCD according to the predicate is_gcd
    ○ Use auxiliary lemma gcd_step_aux, then apply it in the step_a and step_b branches of the induction on gcd.