A BRIEF INTRODUCTION TO (DEPENDENT) TYPE THEORY

Cory Knapp
January 14, 2015
University of Birmingham
WHAT IS TYPE THEORY?
formal language of *terms with types*

\[ x : A \quad \text{“x has type } A \text{”} \]
What is type theory?

formal language of terms with types

$x : A$  “$x$ has type $A$”

Idealized (functional) language

type  datatype.

$x : A$  $x$ is a program (or value) of type $A$. 
What is type theory?

formal language of terms with types

\( x : A \) "\( x \) has type \( A \)"

Idealized (functional) language

type datatype.

\( x : A \) \( x \) is a program (or value) of type \( A \).

\((+): \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\)

\(
\text{map}(+1)[1,2,3]::\text{[Int]}
\)
What is type theory?

formal language of *terms* with *types*

\[ x : A \quad \text{“}x \text{ has type } A\text{”} \]
**What is type theory?**

formal language of *terms with types*

\[ x : A \quad \text{“x has type A”} \]

Logic with book-keeping

\[ x : A \quad A \text{ is a proposition and } x \text{ is a proof of } A. \]
formal language of terms with types

\( x : A \)  

“\( x \) has type \( A \)”

Logic with book-keeping

\( x : A \)  

A is a proposition and \( x \) is a proof of \( A \).

\[ \lambda(x : A) . x : A \rightarrow A \]

Given a proof \( x \) of \( A \), we can find a proof \( x \) of \( A \)
What is type theory?

formal language of terms with types

\[ x : A \quad \text{“}x \text{ has type } A\text{”} \]
What is type theory?

formal language of *terms* with *types*

\[ x : A \quad \text{“} x \text{ has type } A \text{”} \]

Alternative to set theory

\[ x : A \quad A \text{ is a set and } x \in A \]
What is type theory?

formal language of terms with types

\[ x : A \quad \text{“} x \text{ has type } A \text{”} \]

Alternative to set theory

\[ x : A \quad A \text{ is a set and } x \in A \]

\[ 0 : \mathbb{N} \]

\[ 0 \in \mathbb{N} \]
What is type theory?

formal language of terms with types

\[ x : A \quad \text{“x has type A”} \]
Everything happens in a context

environment — implicitly bound identifiers.
Everything happens in a context environment — implicitly bound identifiers.

Object foo(int x, int y){
    //...
    int z = x + y;
    //...
}

\[ x : \text{int}, y : \text{int} \vdash z : \text{int} \]
Everything happens in a context

environment — implicitly bound identifiers.

Object foo(int x, int y){
    //...
    int z = x + y;
    //...
}

\[ x : \text{int}, y : \text{int} \vdash z : \text{int} \]
Everything happens in a context

environment — implicitly bound identifiers.

Object foo(int x, int y){
  //...
  int z = x + y;
  //...;
}

\(x: \text{int}, y: \text{int} \vdash z: \text{int}\)
Judgment = Result
Judgment = Result

Typing judgments
Judgment = Result

Typing judgments

\[ \Gamma \vdash A : \text{Type} \]
\[ \Gamma \vdash x : A \]
Judgment = Result

Typing judgments
   \( \Gamma \vdash A : \text{Type} \)
   \( \Gamma \vdash x : A \)

Equality judgments
Rules and Judgments

Judgment = Result

Typing judgments
\[ \Gamma \vdash A : \text{Type} \]
\[ \Gamma \vdash x : A \]

Equality judgments
\[ \Gamma \vdash A \equiv B : \text{Type} \]
\[ \Gamma \vdash x \equiv y : A \]
Rules and Judgments

Judgment = Result

Typing judgments
\[ \Gamma \vdash A : \text{Type} \]
\[ \Gamma \vdash x : A \]

Equality judgments
\[ \Gamma \vdash A \equiv B : \text{Type} \]
\[ \Gamma \vdash x \equiv y : A \]
Rules and Judgments

Judgment = Result

Typing judgments
\[ \Gamma \vdash A : \text{Type} \]
\[ \Gamma \vdash x : A \]

Equality judgments
\[ \Gamma \vdash A \equiv B : \text{Type} \]
\[ \Gamma \vdash x \equiv y : A \]

Rule = Step

formally:
\[ \frac{I_1 \quad \ldots \quad I_n}{I} \]
RULE
Judgment = Result

Typing judgments

\[ \Gamma \vdash A : \text{Type} \]

\[ \Gamma \vdash x : A \]

Equality judgments

\[ \Gamma \vdash A \equiv B : \text{Type} \]

\[ \Gamma \vdash x \equiv y : A \]

Rule = Step

formally:

\[
\begin{array}{c}
\mathcal{J}_1 \\
\ldots \\
\mathcal{J}_n \\
\hline
\mathcal{J} \end{array}
\]

RULE

conditional

\[
\Gamma \vdash b : \text{bool} \quad \Gamma \vdash c_1 : C \quad \Gamma \vdash c_2 : C
\]

\[ \Gamma \vdash \text{if } b \text{ then } c_1 \text{ else } c_2 : C \]
**Rules and Judgments**

**Judgment = Result**

**Typing judgments**
- $\Gamma \vdash A : \text{Type}$
- $\Gamma \vdash x : A$

**Equality judgments**
- $\Gamma \vdash A \equiv B : \text{Type}$
- $\Gamma \vdash x \equiv y : A$

**Rule = Step**

**formally:**

\[
\begin{array}{c}
J_1 \\
\vdots \\
J_n
\end{array}
\]

**RULE**

**conditional**

\[
\Gamma \vdash b : \text{bool} \quad \Gamma \vdash c_1 : C \quad \Gamma \vdash c_2 : C
\]

\[
\Gamma \vdash \text{if } b \text{ then } c_1 \text{ else } c_2 : C
\]

**Type isn’t a type (but we can pretend)**
SIMPLE TYPES
Types are defined by rules

Formation rules  When does the type exist?
Types are defined by rules

**Formation rules**  When does the type exist?

\[
\Gamma \vdash A : \text{Type} \quad \Gamma \vdash B : \text{Type} \\
\therefore \Gamma \vdash A \rightarrow B : \text{Type}
\]

function type
Types are defined by rules

**Formation rules**  When does the type exist?

**Introduction rules**  What are the basic terms?
Defining types

Types are defined by rules

Formation rules  When does the type exist?
Introduction rules  What are the basic terms?

\[
\Gamma \vdash B : \text{Type} \quad \Gamma, x : A \vdash b : B \\
\Gamma \vdash \lambda x.b : A \to B
\]

lambda abstraction
Defining types

Types are defined by rules

Formation rules  When does the type exist?
Introduction rules  What are the basic terms?
Elimination rules  What is the basic way to use a term?
Types are defined by rules

Formation rules  When does the type exist?
Introduction rules  What are the basic terms?
Elimination rules  What is the basic way to use a term?

\[
\Gamma \vdash m : A \quad \Gamma \vdash f : A \rightarrow B \\
\Gamma \vdash f(m) : B
\]

application
Types are defined by rules

**Formation rules**  When does the type exist?
**Introduction rules**  What are the basic terms?
**Elimination rules**  What is the basic way to use a term?
**Computation rules**  How do we reduce expressions?
Defining types

Types are defined by rules

Formation rules  When does the type exist?
Introduction rules  What are the basic terms?
Elimination rules  What is the basic way to use a term?
Computation rules  How do we reduce expressions?

\[
\Gamma \vdash B : \text{Type} \quad \Gamma, x : A \vdash b : B \quad \Gamma \vdash m : A
\]

\[
\Gamma \vdash (\lambda x.b)(m) \equiv b[x/m] : B
\]

substitution
Inductive types

Introduction rules

"Constructors"

Elimination/Computation rules

"Pattern-matching"

data List = Cons Int List | Empty

sum :: List -> Int

sum (Cons i l) = i + sum l

sum Empty = 0
Inductive types

Introduction rules “Constructors”
Introduction rules  “Constructors”
Elimination/Computation rules  “Pattern-matching”

data List = Cons Int List | Empty

sum :: List -> Int
sum (Cons i l) = i + sum l
sum Empty = 0
**INDUCTIVE TYPES**

Introduction rules “Constructors”

Elimination/Computation rules “Pattern-matching”

```
data List = Cons Int List | Empty

sum :: List -> Int
sum (Cons i l) = i + sum l
sum Empty = 0
```
Introduction rules "Constructors"
Elimination/Computation rules "Pattern-matching"

data List = Cons Int List | Empty

sum :: List -> Int
sum (Cons i l) = i + sum l
sum Empty = 0
Introduction rules “Constructors”
Elimination/Computation rules “Pattern-matching”

data List = Cons Int List | Empty

sum :: List -> Int
sum (Cons i l) = i + sum l
sum Empty = 0
Introduction rules “Constructors”
Elimination/Computation rules “Pattern-matching”

\[
data \text{ List } = \text{ Cons Int List } | \text{ Empty}
\]

\[
\text{sum :: List } \rightarrow \text{ Int}
\]
\[
\text{sum (Cons i l) } = i + \text{ sum l}
\]
\[
\text{sum Empty } = 0
\]
INDUCTIVE TYPES

Introduction rules “Constructors”
Elimination/Computation rules “Pattern-matching”

data List = Cons Int List | Empty

sum :: List -> Int
sum (Cons i l) = i + sum l
sum Empty = 0
The simple types

\[ A \times B \ (a, b) \]

“Pair A B”
The simple types

\[ A \times B \ (a, b) \]
\[ A + B \ \text{in} \text{la}; \text{inrb} \]

“Pair A B”

“Inl A | Inr B”
The simple types

\[ A \times B \ (a, b) \]
\[ A + B \ \text{inl} a; \text{inr} b \]
empty no constructors!

"Pair A B"
"Inl A | Inr B"
The simple types

\[
A \times B \ (a, b) \\
A + B \ \text{inl}a; \ \text{inrb} \\
\text{empty} \ \text{no constructors!} \\
\text{unit} \ \star
\]

“Pair A B”
“Inl A | Inr B”

()
The simple types

\[ A \times B \ (a, b) \]

\[ A + B \ \text{inla}; \ \text{inrb} \]

empty \ no constructors!

unit \ \star \]

others? \ Bool, \ Int, \ Nat, \ well-founded \ trees...
<table>
<thead>
<tr>
<th>Types</th>
<th>Sets</th>
<th>Propositions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What about quantifiers?
<table>
<thead>
<tr>
<th>Types</th>
<th>Sets</th>
<th>Propositions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \times B$</td>
<td>$A \times B$</td>
<td>$A \land B$</td>
</tr>
</tbody>
</table>
## Interpretation

<table>
<thead>
<tr>
<th>Types</th>
<th>Sets</th>
<th>Propositions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \times B$</td>
<td>$A \times B$</td>
<td>$A \land B$</td>
</tr>
<tr>
<td>$A + B$</td>
<td>$A \cup B$</td>
<td>$A \lor B$</td>
</tr>
<tr>
<td>Types</td>
<td>Sets</td>
<td>Propositions</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td>$A \times B$</td>
<td>$A \times B$</td>
<td>$A \land B$</td>
</tr>
<tr>
<td>$A + B$</td>
<td>$A \cup B$</td>
<td>$A \lor B$</td>
</tr>
<tr>
<td>$A \to B$</td>
<td>$B^A$</td>
<td>$B \to A$</td>
</tr>
</tbody>
</table>
**Interpretation**

<table>
<thead>
<tr>
<th>Types</th>
<th>Sets</th>
<th>Propositions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \times B$</td>
<td>$A \times B$</td>
<td>$A \land B$</td>
</tr>
<tr>
<td>$A + B$</td>
<td>$A \cup B$</td>
<td>$A \lor B$</td>
</tr>
<tr>
<td>$A \rightarrow B$</td>
<td>$B^A$</td>
<td>$B \rightarrow A$</td>
</tr>
<tr>
<td><strong>unit</strong></td>
<td>${\star}$</td>
<td>$\top$</td>
</tr>
</tbody>
</table>
## Interpretation

<table>
<thead>
<tr>
<th>Types</th>
<th>Sets</th>
<th>Propositions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \times B$</td>
<td>$A \times B$</td>
<td>$A \land B$</td>
</tr>
<tr>
<td>$A + B$</td>
<td>$A \cup B$</td>
<td>$A \lor B$</td>
</tr>
<tr>
<td>$A \to B$</td>
<td>$B^A$</td>
<td>$B \to A$</td>
</tr>
<tr>
<td>unit</td>
<td>${\star}$</td>
<td>$\top$</td>
</tr>
<tr>
<td>empty</td>
<td>$\emptyset$</td>
<td>$\bot$</td>
</tr>
<tr>
<td>Types</td>
<td>Sets</td>
<td>Propositions</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------</td>
<td>---------------</td>
</tr>
<tr>
<td>$A \times B$</td>
<td>$A \times B$</td>
<td>$A \land B$</td>
</tr>
<tr>
<td>$A + B$</td>
<td>$A \cup B$</td>
<td>$A \lor B$</td>
</tr>
<tr>
<td>$A \rightarrow B$</td>
<td>$B^A$</td>
<td>$B \rightarrow A$</td>
</tr>
<tr>
<td>unit</td>
<td>${\star}$</td>
<td>$\top$</td>
</tr>
<tr>
<td>empty</td>
<td>$\emptyset$</td>
<td>$\bot$</td>
</tr>
</tbody>
</table>

What about quantifiers?
DEPENDENT TYPES
Dependent types

Γ, x : A ⊢ B : Type
Dependent types

\[ \Gamma, x : A \vdash B : \text{Type} \]

\( B \) depends on \( x : A \)
Dependent types

\[ \Gamma, x : A \vdash B : \text{Type} \]

*B depends on* \( x : A \)

Vectors of length \( n : \text{nat} \).
Days of month \( m : \text{Months} \).
Dependent types

\[ \Gamma, x : A \vdash B : \text{Type} \]

*B depends on* \(x : A\)

Vectors of length \(n : \text{nat}\).
Days of month \(m : \text{Months}\).

\(\lambda\)-abstraction:

\[ \Gamma \vdash \lambda x. B : A \to \text{Type} \]

A type *family*
Dependent products

dependent functions

\[ \text{lastDay} : \text{daysOf} \]

\[ \prod_{x \in A} B \]

\[ \text{lastDay} : \prod_{m \in \text{Month}} \text{daysOf}(m) \]
Dependent functions

Output type depends on input.
dependent functions

Output type depends on input.

lastDay(January) : daysOf(January)
lastDay(April) : daysOf(April)
Dependent products

dependent functions

Output type depends on input.

lastDay(January) : daysOf(January)
lastDay(April) : daysOf(April)

\( \Pi_{x:A} B \)

dependent product
Dependent products

dependent functions

Output type depends on input.

lastDay(January) : daysOf(January)
lastDay(April) : daysOf(April)

\[ \prod_{x:A} B \]

lastDay : \[ \prod_{m:Month} \text{daysOf}(m) \]

dependent product
Dependent functions are generalized functions

generalize elimination
Dependent functions are generalized functions

generalize elimination

\[ C : A \times B \rightarrow \text{Type} \]

\[ f : \prod_{p : A \times B} C(p) \]

\[ f((a, b)) := \text{something} \]
Dependent functions are generalized functions

generalize elimination

\[ C : A \times B \rightarrow \text{Type} \]

\[ f : \prod_{p : A \times B} C(p) \]
\[ f((a, b)) := \text{something} \]

*Induction principle*
dependent pair

second coordinate depends on first
Dependent sums

dependent pair

second coordinate depends on first

red-black tree: binary tree $T$ with color assignment $f : \text{color}(T)$
dependent pair

second coordinate depends on first

red-black tree: binary tree $T$ with color assignment $f : \text{color}(T)$

$(T, f)$
dependent pair

second coordinate depends on first

red-black tree: binary tree $T$ with color assignment $f : \text{color}(T)$

$$(T, f)$$

$$\sum_{x:A} B$$

dependent sum
Dependent pair

second coordinate depends on first

red-black tree: binary tree $T$ with color assignment $f : color(T)$

$$(T,f)$$

$$\sum_{x:A} B$$

$$\text{RBTree} := \sum_{T: \text{BinTree}} color(T)$$

dependent sum
<table>
<thead>
<tr>
<th>Types</th>
<th>Sets</th>
<th>Propositions</th>
</tr>
</thead>
</table>

\[
\prod_{x:A} B(x) \quad \Sigma_{x:A} B(x)
\]

**Interpretation**
Types

\( B : A \rightarrow \text{Type} \)

Sets

\( \{ B_a \mid a \in A \} \)

Propositions

predicate \( B \) on \( A \)
### Types

\[ B : A \rightarrow \text{Type} \]
\[ \prod_{x:A} B(x) \]

### Sets

\[ \{ B_a \mid a \in A \} \]
\[ \prod_{x:A} B(x) \]

### Propositions

Predicate \( B \) on \( A \)
\[ \forall x \in A(B(x)) \]
Types

$B : A \rightarrow \text{Type}$

$\prod_{x : A} B(x)$

$\sum_{x : A} B(x)$

Sets

$\{B_a \mid a \in A\}$

$\prod_{x : A} B(x)$

$\biguplus_{x : A} B(x)$

Propositions

predicate $B$ on $A$

$\forall x \in A(B(x))$

$\exists x \in A(B(x))$
Propositions are types
\( x = y \) is a proposition.

Identity types
Propositions are types
$x = y$ is a proposition.

Identity types

Inductive \textit{family} $=_A : A \to A \to \text{Type}$
Propositions are types
\(x = y\) is a proposition.

Identity types

Inductive family \(\equiv_A: A \to A \to \text{Type}\)

Constructor \(\text{refl}_a : a =_A a\)
Propositions are types
\( x = y \) is a proposition.

Identity types

Inductive family \( \equiv_A : A \to A \to \text{Type} \)
Constructor \( \text{refl}_a : a \equiv_A a \)
Eliminate by pattern-match on \( \text{refl} \)
Propositions are types
$x = y$ is a proposition.

Identity types

Inductive family $=_A : A \to A \to \text{Type}$
Constructor $\text{refl}_a : a =_A a$
Eliminate by pattern-match on $\text{refl}$

\[
\text{sym} : \prod_{x,y:X}(x = y \to y = x) \\
\text{sym}_{x,x}(\text{refl}_x) = \text{refl}_x
\]
Is $\equiv$ the same as $=$?
TWO NOTIONS OF EQUALITY

Is $\equiv$ the same as $=$?

Extensional
Two notions of equality

Is $\equiv$ the same as $=$?

Extensional

Intensional
Two notions of equality

Is $\equiv$ the same as $=$?

Extensional
Too rigid

Intensional
TWO NOTIONS OF EQUALITY

Is $\equiv$ the same as $=$?

Extensional
Too rigid

Intensional
Too flexible
TWO NOTIONS OF EQUALITY

Is \( \equiv \) the same as \( = \)?

Extensional
Too rigid

Intensional
Too flexible

Can we find a middle ground?
HOMOTOPY TYPE THEORY
Homotopy interpretation

Type theory: formal language for homotopy theory

Voevodsky (2006); Awodey & Warren (2007)
Type theory: formal language for homotopy theory

Voevodsky (2006); Awodey & Warren (2007)

- types are spaces
- terms are points
- equalities are paths
Type theory: formal language for homotopy theory

Voevodsky (2006); Awodey & Warren (2007)

types are spaces
terms are points
equalities are paths

homotopy theory

Use paths to characterize spaces
Natural interpretations
homotopy theory

Equivalent spaces are identified

$A \simeq B$ if there is an invertible function $A \to B$
homotopy theory

Equivalent spaces are identified

\( A \simeq B \) if there is an invertible function \( A \to B \)

Univalence Axiom

\[(A \simeq B) \simeq (A =_{\text{Type } B})\]
Paths are part of a space.
Paths are part of the definition of a space.
Paths are part of the definition of a space.

Allow *path constructors*.
Paths are part of the definition of a space.

Allow $path$ constructors.

Higher-inductive types
Paths are part of the definition of a space.

Allow \textit{path} constructors.

\textbf{Higher-inductive types}

Point constructors $x : A$
Paths are part of the definition of a space.

Allow path constructors.

Higher-inductive types

Point constructors $x : A$
path constructors $p : x =_A y$
Paths are part of the definition of a space.

Allow *path* constructors.

**Higher-inductive types**

Point constructors $x : A$

path constructors $p : x \equiv_A y$

Quotients, pushouts, suspensions, more.

github.com/HoTT—Computer implementations

Questions?