A BRIEF INTRODUCTION TO (DEPENDENT) TYPE THEORY

Cory Knapp

January 14, 2015

University of Birmingham

WHAT IS TYPE THEORY?

What is type theory?

formal language of *terms* with *types*

$x : A$ "x has 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.

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.

```
(+) :: Int -> Int -> Int
map (+1) [1,2,3] :: [Int]
```

What is type theory?

formal language of *terms* with *types*

$x : A$ "x has type A"

What is type theory?

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.

What is type theory?

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.

$$\lambda(x : A)\ .\ x : A \to A$$

Given a proof x of A, we can find a proof x of A

What is type theory?

formal language of *terms* with *types*

$x : A$ "x has type A"

What is type theory?

formal language of *terms* with *types*

$x : A$ "x has type A"

Alternative to set theory

$x : A$ A is a set and $x \in A$

What is type theory?

formal language of *terms* with *types*

$x : A$ "x has type A"

Alternative to set theory

$x : A$ A is a set and $x \in A$

$$0 : \mathbb{N}$$

$$0 \in \mathbb{N}$$

What is type theory?

formal language of *terms* with *types*

$x : A$ "x has type A"

Contexts

Everything happens in a context

environment — implicitly bound identifiers.

Contexts

Everything happens in a context

environment — implicitly bound identifiers.

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

$$x : \mathtt{int}, y : \mathtt{int} \vdash z : \mathtt{int}$$

Contexts

Everything happens in a context

environment — implicitly bound identifiers.

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

$$x : \texttt{int}, y : \texttt{int} \vdash z : \texttt{int}$$

Contexts

Everything happens in a context

environment — implicitly bound identifiers.

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

$$x : \texttt{int}, y : \texttt{int} \vdash z : \texttt{int}$$

Rules and judgments

Judgment = Result

Rules and judgments

Judgment = Result

Typing judgments

Rules and judgments

Judgment = Result

Typing judgments

$\Gamma \vdash A : \mathsf{Type}$

$\Gamma \vdash x : A$

Rules and judgments

Judgment = Result

Typing judgments

$\Gamma \vdash A : \mathsf{Type}$

$\Gamma \vdash x : A$

Equality judgments

Rules and judgments

Judgment = Result

Typing judgments

$\Gamma \vdash A : \mathsf{Type}$

$\Gamma \vdash x : A$

Equality judgments

$\Gamma \vdash A \equiv B : \mathsf{Type}$

$\Gamma \vdash x \equiv y : A$

Rules and judgments

Judgment = Result

Rule = Step

Typing judgments

$\Gamma \vdash A : \mathsf{Type}$

$\Gamma \vdash x : A$

Equality judgments

$\Gamma \vdash A \equiv B : \mathsf{Type}$

$\Gamma \vdash x \equiv y : A$

Rules and judgments

Judgment = Result

Typing judgments

$\Gamma \vdash A : \mathsf{Type}$

$\Gamma \vdash x : A$

Equality judgments

$\Gamma \vdash A \equiv B : \mathsf{Type}$

$\Gamma \vdash x \equiv y : A$

Rule = Step

formally:

$$\frac{\mathcal{J}_1 \qquad \dots \qquad \mathcal{J}_n}{\mathcal{J}} \ \text{RULE}$$

Rules and judgments

Judgment = Result

Typing judgments

$\Gamma \vdash A : \texttt{Type}$

$\Gamma \vdash x : A$

Equality judgments

$\Gamma \vdash A \equiv B : \texttt{Type}$

$\Gamma \vdash x \equiv y : A$

Rule = Step

formally:

$$\frac{\mathcal{J}_1 \qquad \ldots \qquad \mathcal{J}_n}{\mathcal{J}}\ \text{RULE}$$

conditional

$$\frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash c_1 : C \quad \Gamma \vdash c_2 : C}{\Gamma \vdash \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \texttt{:}\ C}$$

Rules and judgments

Judgment = Result

Typing judgments

$\Gamma \vdash A : \mathtt{Type}$

$\Gamma \vdash x : A$

Equality judgments

$\Gamma \vdash A \equiv B : \mathtt{Type}$

$\Gamma \vdash x \equiv y : A$

Rule = Step

formally:

$$\frac{\mathcal{J}_1 \qquad \dots \qquad \mathcal{J}_n}{\mathcal{J}}\ \text{RULE}$$

conditional

$$\frac{\Gamma \vdash b : \mathtt{bool} \quad \Gamma \vdash c_1 : C \quad \Gamma \vdash c_2 : C}{\Gamma \vdash \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\ \mathtt{:}\ C}$$

`Type` isn't a type (but we can pretend)

SIMPLE TYPES

Defining types

Types are defined by rules

Formation rules When does the type exist?

Defining types

Types are defined by rules

Formation rules When does the type exist?

$$\frac{\Gamma \vdash A : \mathtt{Type} \qquad \Gamma \vdash B : \mathtt{Type}}{\Gamma \vdash A \to B : \mathtt{Type}}$$

function type

Defining types

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?

$$\frac{\Gamma \vdash B : \mathtt{Type} \qquad \Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : A \to B}$$

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?

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?

$$\frac{\Gamma \vdash m : A \qquad \Gamma \vdash f : A \to B}{\Gamma \vdash f(m) : B}$$

application

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?

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?

$$\frac{\Gamma \vdash B : \mathtt{Type} \qquad \Gamma, x : A \vdash b : B \qquad \Gamma \vdash m : A}{\Gamma \vdash (\lambda x.b)(m) \equiv b[x/m] : B}$$

substitution

Inductive types

Inductive types

Introduction rules "Constructors"

Inductive types

Introduction rules “Constructors”

Elimination/Computation rules “Pattern-matching”

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"

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
```

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”

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) "Pair A B"

$A + B$ $\mathtt{inl}\, a; \mathtt{inr}\, b$ "Inl A | Inr B"

The simple types

$A \times B$	(a, b)	“`Pair A B`”
$A + B$	$\texttt{inl}\,a$; $\texttt{inr}\,b$	“`Inl A	Inr B`”
`empty`	no constructors!	

The simple types

$A \times B$	(a, b)	“`Pair A B`”
$A + B$	$\texttt{inl}\,a; \texttt{inr}\,b$	“`Inl A | Inr B`”
`empty`	no constructors!	
`unit`	$\star$	`()`

The simple types

$A \times B$	(a, b)	“`Pair A B`”
$A + B$	`inl` a; `inr` b	“`Inl A	Inr B`”
`empty`	no constructors!	
`unit`	$\star$	`()`
others?	`Bool`, `Int`, `Nat`, well-founded trees...	

Interpretation

Types | Sets | Propositions

Interpretation

Types	Sets	Propositions
$A \times B$	$A \times B$	$A \wedge B$

Interpretation

Types	Sets	Propositions
$A \times B$	$A \times B$	$A \wedge B$
$A + B$	$A \cup B$	$A \vee B$

Interpretation

Types	Sets	Propositions
$A \times B$	$A \times B$	$A \wedge B$
$A + B$	$A \cup B$	$A \vee B$
$A \to B$	B^A	$B \to A$

Interpretation

Types	Sets	Propositions
$A \times B$	$A \times B$	$A \wedge B$
$A + B$	$A \cup B$	$A \vee B$
$A \to B$	B^A	$B \to A$
$\texttt{unit}$	$\{\star\}$	$\top$

Interpretation

Types	Sets	Propositions
$A \times B$	$A \times B$	$A \wedge B$
$A + B$	$A \cup B$	$A \vee B$
$A \to B$	B^A	$B \to A$
`unit`	$\{\star\}$	$\top$
`empty`	$\emptyset$	$\bot$

Interpretation

Types	Sets	Propositions
$A \times B$	$A \times B$	$A \wedge B$
$A + B$	$A \cup B$	$A \vee B$
$A \to B$	B^A	$B \to A$
`unit`	$\{\star\}$	$\top$
`empty`	$\emptyset$	$\bot$

What about quantifiers?

DEPENDENT TYPES

Dependent types

$$\Gamma, x : A \vdash B : Type$$

Dependent types

$$\Gamma, x : A \vdash B : \mathit{Type}$$

B *depends* on $x : A$

Dependent types

$$\Gamma, x : A \vdash B : \mathit{Type}$$

B depends on $x : A$

Vectors of length n : `nat`.
Days of month m : Months.

Dependent types

$$\Gamma, x : A \vdash B : Type$$

B depends on $x : A$

Vectors of length n : `nat`.
Days of month m : Months.

λ-abstraction:

$\Gamma \vdash \lambda x.B : A \to \mathtt{Type}$

A type family

Dependent products

dependent functions

Dependent products

dependent functions

Output type depends on input.

Dependent products

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)
```

$\prod_{x:A} B$

dependent product

Dependent products

dependent functions

Output type depends on input.

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

$\prod_{x:A} B$

$\texttt{lastDay} : \prod_{m:Month} \texttt{daysOf}(m)$

dependent product

Dependent elimination

Dependent functions are generalized functions

generalize elimination

Dependent elimination

Dependent functions are generalized functions

generalize elimination

$C : A \times B \to \texttt{Type}$

$f : \prod_{p:A\times B} C(p)$
$f((a,b)) :=$ *something*

Dependent elimination

Dependent functions are generalized functions

generalize elimination

$C : A \times B \to \mathtt{Type}$

$f : \prod_{p:A\times B} C(p)$
$f((a,b)) :=$ *something*

Induction principle

Dependent sums

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 $f : color(T)$

Dependent sums

dependent pair

second coordinate depends on first

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

$$(T, f)$$

Dependent sums

dependent pair

second coordinate depends on first

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

(T, f)

$\sum_{x:A} B$

dependent sum

Dependent sums

dependent pair

second coordinate depends on first

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

$$(T, f)$$

$$\textstyle\sum_{x:A} B$$
$$\texttt{RBTree} := \textstyle\sum_{T:\texttt{BinTree}} \mathit{color}(T)$$

dependent sum

Interpretation

Types Sets Propositions

Interpretation

Types	Sets	Propositions
$B : A \to \texttt{Type}$	$\{B_a \mid a \in A\}$	predicate B on A

Interpretation

Types

$B : A \to \mathtt{Type}$

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

Sets

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

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

Propositions

predicate B on A

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

Interpretation

Types

$B : A \to \mathtt{Type}$
$\prod_{x:A} B(x)$
$\sum_{x:A} B(x)$

Sets

$\{B_a \mid a \in A\}$
$\prod_{x:A} B(x)$
$\biguplus_{x:A} B(x)$

Propositions

predicate B on A
$\forall x \in A(B(x))$
$\exists x \in A(B(x))$

Equality

Propositions are types

$x = y$ is a proposition.

Identity types

Equality

Propositions are types

$x = y$ is a proposition.

Identity types

Inductive family $=_A : A \to A \to \texttt{Type}$

Equality

Propositions are types

$x = y$ is a proposition.

Identity types

Inductive family $=_A : A \to A \to \texttt{Type}$

Constructor $\texttt{refl}_a : a =_A a$

Equality

Propositions are types

$x = y$ is a proposition.

Identity types

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

Equality

Propositions are types

$x = y$ is a proposition.

Identity types

Inductive family $=_A : A \to A \to \mathtt{Type}$

Constructor $\mathtt{refl}_a : a =_A a$

Eliminate by pattern-match on $\mathtt{refl}$

$$\mathtt{sym} : \prod_{x,y:X} (x = y \to y = x)$$

$$\mathtt{sym}_{x,x}(\mathtt{refl}_x) = \mathtt{refl}_x$$

Two notions of equality

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

Voevdsky (2006); Awodey & Warren (2007)

Homotopy interpretation

Type theory: formal language for homotopy theory

Voevdsky (2006); Awodey & Warren (2007)

- types are spaces
- terms are points
- equalities are paths

Homotopy interpretation

Type theory: formal language for homotopy theory

Voevdsky (2006); Awodey & Warren (2007)

- types are spaces
- terms are points
- equalities are paths

homotopy theory

- Use paths to characterize spaces
- Natural interpretations

UNIVALENCE

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

$$(A \simeq B) \simeq (A =_{\mathsf{Type}} B)$$

Higher Inductive types

Paths are part of a space.

Higher Inductive types

Paths are part of the definition of a space.

Higher Inductive types

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 *path* constructors.

Higher-inductive types

Higher Inductive types

Paths are part of the definition of a space.

Allow *path* constructors.

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 $p : x =_A y$

Higher Inductive types

Paths are part of the definition of a space.

Allow *path* constructors.

Higher-inductive types

Point constructors $x : A$

path constructors $p : x =_A y$

Quotients, pushouts, suspensions, more.

More Information

`homotopytypetheory.org`—Blog, book, articles.

`github.com/HoTT`—Computer implementations

Questions?