Introduction to Dependent Type Theory

Outline

- Refresher on non-dependent type theory
- Motivation for dependent type theory
- Formulation of a very simple type theory
- Me trying to convince you that it's related to programming, with words
- Me trying to convince you that it's related to programming, with an underprepared demo

Non-dependent type theory

- Starting with propositional logic
  - → atoms, connectives, LEM
  - → principled presentation: natural deduction

$\Gamma \vdash A$ "proposition A holds in context Γ"

"context" ~ list of propositions assumed to hold

proposition

  - → technically also $\Gamma \vdash A \ \mathrm{prop}$ for "'A' denotes a proposition in context Γ", but often left implicit in treatments of non-dependent type theories

Connective rules

→ "formation" – when does a symbol "represent" a proposition? (omitted)

"introduction" – when do we know that a proposition holds?

"elimination" – what's a "natural" way to use a proposition?

Conjunction

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge I$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge E_1 \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge E_2$$

Disjunction

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee I_1 \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee I_2$$

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C} \vee E$$

True

$$\frac{}{\Gamma \vdash \top} \top I$$

False

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A} \bot E$$

Connective rules

Implication

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to I$$

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B} \to E$$

Negation

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A} \neg I$$

$$\frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash \bot} \neg E$$

Odd one out

Excluded middle

$$\frac{}{\Gamma \vdash A \vee \neg A}\ LEM$$

→ doesn't eliminate anything

→ introduces a disjunction and/or negation, but we already have "natural" rules for those

→ let's ignore it for now

- Flavors of logic without LEM are called "constructive" or "intuitionistic"
- Has the right vibe for everyday programming: *waves hands*

conjunctions ~ tuples
disjunctions ~ discriminated unions
implications ~ functions

⇒ can we make this precise?

From propositions to types

- When constructing a proof tree, we can build up a string of "tags" to encode the proof
- Such strings are called programs / terms
- Instead of just "a proposition A holds", we may be more specific and say "a program p proves A"
- We don't regard propositions as contentless true/false statements, but as a collection of programs proving a given proposition

$\rightsquigarrow$ judgments $\quad \Gamma \vdash t : A \quad$ "term t has type A in context Γ"

Γ: context $\sim$ list of variables $x : B$

t: term

A: type

Simply typed λ-calculus

→ tagging the introduction & elimination rules

$$\frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}{\Gamma \vdash (s, t) : A \times B}$$

$$\frac{\Gamma \vdash s : A}{\Gamma \vdash \mathrm{inl}(s) : A + B} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathrm{inr}(t) : A + B}$$

$$\frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathrm{pr}_1(p) : A} \qquad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathrm{pr}_2(p) : B}$$

$$\frac{\Gamma \vdash p : A + B \qquad \Gamma, x : A \vdash s : C \qquad \Gamma, x : B \vdash t : C}{\Gamma \vdash \mathrm{case}\ p\ \mathrm{of}\ \{\mathrm{inl}(x) \to s\, ;\ \mathrm{inr}(x) \to t\} : C}$$

$$\frac{\Gamma, x : A \vdash s : B}{\Gamma \vdash \lambda x{:}A.\, s : A \to B}$$

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash f\, s : B}$$

Computation

— We know that the λ-calculus models computation — where is it?

$$\Gamma \vdash s \doteq_A t$$

"s and t are the same element of A in context Γ"

⟶ computation rules

$$\frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}{\Gamma \vdash pr_1(s,t) \doteq_A s}$$

$$\frac{\Gamma \vdash p : A \qquad \Gamma, x : A \vdash s : C \qquad \Gamma, x : B \vdash t : C}{\Gamma \vdash \mathrm{case}\ p\ \mathrm{of}\ \{inl(x) \to s;\ inr(x) \to t\} \doteq_C s[x \mapsto p]}$$

$$\frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}{\Gamma \vdash pr_2(s,t) \doteq_B t}$$

$$\frac{\Gamma \vdash p : B \qquad \Gamma, x : A \vdash s : C \qquad \Gamma, x : B \vdash t : C}{\Gamma \vdash \mathrm{case}\ p\ \mathrm{of}\ \{inl(x) \to s;\ inr(x) \to t\} \doteq_C t[x \mapsto p]}$$

$$\frac{\Gamma, x : A \vdash s : B \qquad \Gamma \vdash t : A}{\Gamma \vdash (\lambda x : A.\ s)\ t \doteq_B s[x \mapsto t]}$$

Computation

- what's the analogue of computation on the logic side?

  $\Rightarrow$ "proof reduction" e.g.

$$\dfrac{\dfrac{\begin{matrix}\vdots \\ \Gamma \vdash A\end{matrix} \quad \begin{matrix}\vdots \\ \Gamma \vdash B\end{matrix}}{\Gamma \vdash A \wedge B}}{\Gamma \vdash A} \quad \doteq \quad \begin{matrix}\vdots \\ \Gamma \vdash A\end{matrix}$$

- Problem: we have no way to talk about behavior in the language

  $\rightsquigarrow$ time to pass over to higher order logic / dependent type theory!

Sidenote – FOL

→ FOL allows us to quantify over objects in a "domain of discourse"

→ problems :

- there is only one domain

$$\left.\begin{array}{l}\forall x.\, N(x) \Rightarrow \varphi \\ \exists x.\, N(x) \land \psi\end{array}\right\}$$ "I only want to talk about Nats"

- we can only quantify over the domain

"induction on natural numbers is applicable to every property"

! not expressible

→ fixing those, we arrive at some basic dependent type theory

Dependent type theory

- We make the language richer to allow ourselves to talk about programs

- Give types access to contexts: $\Gamma \vdash A$ type
  - $\rightsquigarrow$ A is a dependent type, because it may refer to variables
  - $\rightsquigarrow$ Contexts are no longer simple lists
    - — we call them "telescopes"

Ex $x : A,\ y : B,\ z : C \vdash D$ type

B: may refer to x

C: may refer to x, y

D: may refer to x, y, z

Quantification

Π types for "forall", Σ types for "exists"

$$\frac{\Gamma, x : A \vdash B \text{ type}}{\Gamma \vdash \Pi(x:A).B \text{ type}}$$

$$\frac{\Gamma, x : A \vdash s : B}{\Gamma \vdash \lambda x{:}A.\,s : \Pi(x:A).B}$$

$$\frac{\Gamma \vdash f : \Pi(x:A).B \qquad \Gamma \vdash s : A}{\Gamma \vdash f\,s : B[x \mapsto s]}$$

$$\frac{\Gamma, x : A \vdash B \text{ type}}{\Gamma \vdash \Sigma(x:A).B \text{ type}}$$

$$\frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B[x \mapsto s]}{\Gamma \vdash s, t : \Sigma(x:A).B}$$

$$\frac{\Gamma \vdash p : \Sigma(x:A).B}{\Gamma \vdash pr_1\, p : A}$$

$$\frac{\Gamma \vdash p : \Sigma(x:A).B}{\Gamma \vdash pr_2\, p : B[x \mapsto pr_1\, p]}$$

Terminology

- Π and Σ types are named after counting their inhabitants: for a type A of size $|A|$, and a dependent type B over A (so $x : A \vdash B$ type)

  we have $|\Pi(x:A).\,B| = \text{"}\prod_{x \in A} |B\,x|\text{"}$ ← product

  and $|\Sigma(x:A).\,B| = \text{"}\sum_{x \in A} |B\,x|\text{"}$ ← sum

- Confusingly enough, we call the type $\Pi(x:A).\,B$ a "dependent product", but its inhabitants $f : \Pi(x:A).\,B$ "dependent functions"
- Similarly, the type $\Sigma(x:A).\,B$ is a "dependent sum", but its inhabitants $(a, b) : \Sigma(x:A).\,B$ are "dependent pairs"

Universes

- the judgement "$\Gamma \vdash A$ type" is on the meta-level, and we want to "internalize" it

→ we introduce the type of (some) types Type

→ "$\Gamma \vdash A$ type" becomes "$\Gamma \vdash A : \mathrm{Type}$"

→ dependent types are just regular functions into Type:
"$\Gamma, x : A \vdash B$ type" becomes "$\Gamma \vdash B : A \to \mathrm{Type}$"

→ consistency prevents us from having $\mathrm{Type} : \mathrm{Type}$,
so we usually build a hierarchy: $\mathrm{Type} : \mathrm{Type}_1,\ \mathrm{Type}_1 : \mathrm{Type}_2, \ldots$

Identity types

- Very interesting!

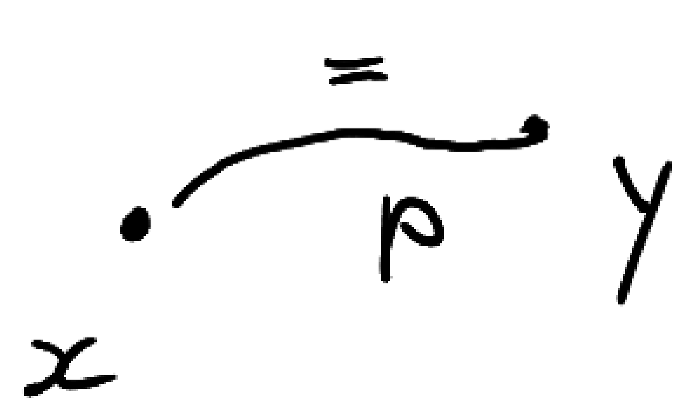
- $\Gamma, x : A, y : A \vdash x =_A y : \mathrm{Type}$ $\qquad$ $\Gamma, x : A \vdash \mathrm{refl}_x : x =_A x$

- induction

$$\frac{\begin{array}{l}\Gamma \vdash C : \Pi (x, y : A) (p : x =_A y).\ \mathrm{Type} \\ \Gamma \vdash c : \Pi (x : A).\ C(x, x, \mathrm{refl}_x)\end{array}}{\Gamma, x : A, y : A, p : x =_A y \vdash J(x, y, p, c) : C(x, y, p)}$$

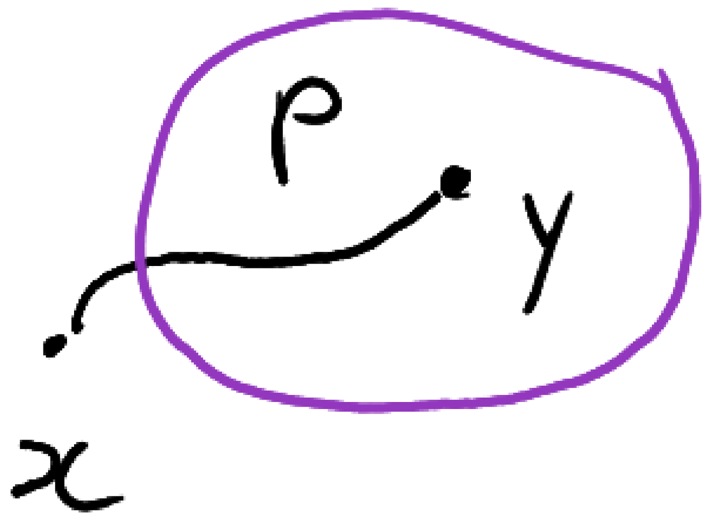
"vacuum cord principle"

if your property varies along a path, then you can contract the path to refl



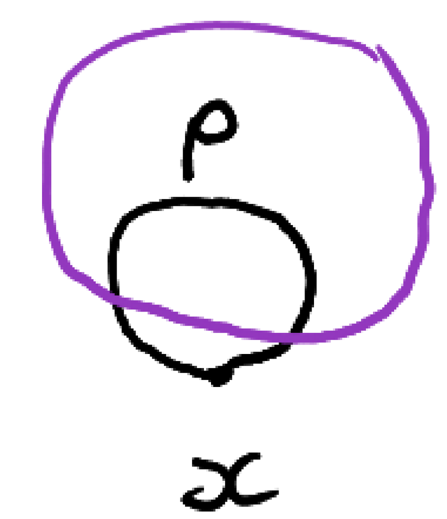
Sidenote: Axiom K

- "why isn't every loop refl"? — the identity type is generated as a pair (endpoint, path-to-endpoint)



this part is generated by refl

→ that's different than saying that



this part is generated by refl

→ we can either leave this ambiguous (Rocq),
or introduce Axiom K (Lean, default Agda)

$$\Gamma, x : A, p : x =_A x \vdash p \doteq \mathrm{refl}$$

or introduce univalence (HoTT, Agda --without-K, cubical)

Inductive types

Ex Nat

$$\frac{}{\Gamma \vdash Nat : Type} \qquad \frac{}{\Gamma \vdash 0 : Nat} \qquad \frac{\Gamma \vdash n : Nat}{\Gamma \vdash suc\ n : Nat}$$

$$\frac{\Gamma \vdash P : Nat \to Type \qquad \Gamma \vdash z : P\,0 \qquad \Gamma, n : Nat \vdash s : P\,n \to P(suc\ n)}{\Gamma, n : Nat \vdash ind_{Nat}(P, z, s, n) : P(n)}$$

$$ind_{Nat}(P, z, s, 0) \doteq z \qquad ind_{Nat}(P, z, s, n+1) \doteq s\ n\ (ind_{Nat}(P, z, s, n))$$

After abstracting the inputs, ind_{Nat} has the type

$$ind_{Nat} : \Pi(P : Nat \to Type).\ P\,0 \to (\Pi(n : Nat).\ P\,n \to P(suc\ n)) \to \underbrace{\Pi(n : Nat).\ P\,n}_{\text{"property } P \text{ holds for all natural numbers"}}$$

Compare

$$\mathrm{ind}_{Nat} : \Pi(P : Nat \to Type).\ P\ 0 \to (\Pi(n:Nat).\ P\ n \to P\ (suc\ n))$$
$$\to \Pi(n : Nat).\ P\ n$$

and

constant $Z/0$
function $S/1$

predicate $Nat/1$

axiom $Nat(Z)$
axiom $\forall n.\ Nat(n) \to Nat(Sn)$

axiom $\mathrm{ind}^{\varphi}_{Nat} : \varphi 0 \to (\forall x.\ Nat(x) \to \varphi n \to \varphi(Sn))$
$$\to \forall n.\ Nat(n) \to \varphi n$$

for every formula $\varphi(x)$

Look what they need to limit a fraction of our power

$Nat(x) \rightsquigarrow$ "typal predicate"

$\varphi(x) \rightsquigarrow$ property, but what if we want a function?

$\to$ functions need to be encoded as graph predicates

Inductive types

There is a general theory to allow user-defined inductive types

→ W types

$$\frac{\Gamma \vdash S : \mathrm{Type} \qquad \Gamma \vdash P : A \to \mathrm{Type}}{\Gamma \vdash W\ S\ P : \mathrm{Type}}$$

S ~ "shapes" $\quad P s$ ~ "positions in shape s"

→ out of scope of this talk!

→ but look them up!

→ related concept: "containers"! $S \triangleleft P$

→ I don't know how they relate yet!

→ but it involves categories!

Proof assistants

- not well-defined, and often the wrong name
- originally dependently typed languages were used just for theorem proving (e.g. Coq since 1989)
- my usage of the term:
  → consider a dependently typed language (e.g. Lean, Agda, Coq)

  → dependent types impose greater restrictions on implementations (that's the point!), it's convenient to write them interactively

  → this interactivity needs to talk to the typechecker, so this interactive layer is what I would call a "proof assistant"

  → NOT a prover, which tries to synthesize proofs by itself