Functional Programming and Proving in Coq

Matthieu Sozeau
<matthieu.sozeau@inria.fr>

“Beware of bugs in the above code; I have only proved it correct, not tried it.”

–Donald Knuth, *Notes on the van Emde Boas construction of priority deques: An instructive use of recursion,*1977

“There are two ways to write error-free programs; only the third one works.”

–Alan Perlis,“Epigrams on Programming”, Turing Award citation,1982

Formal Proof Is The Answer!

- **Mechanically** check program **correctness** with respect to a **logical** specification.
- (Relative) Logical **consistency** ensures the **specification is all you need** to believe…
- …plus ZF(C) or some other trusted foundation, and the implementation of the proof checker.

But No Silver Bullet

“Even perfect program verification can only establish that a program meets its specification. […] Much of the essence of building a program is in fact the *debugging of the specification*. [italics added]”

–Fred Brooks, “The Mythical Man-Month”, 1986

Outline

Yesterday, the specification language and a bit of proofs & programs.

Today, a bit of history/philosophy and then the full GALLINA language of proofs & programs.

Recall Total Functional Programming

- Basically, **effective** mathematical functions (dependently-typed λ-calculus).
  - All functions **must** terminate with a value → no **`eval`** or **`collatz`**
  - Type checking ensures functions cannot lie about what they are doing, or hide any side-effect. **You can trust types**. (Typing is noted $p : A \to B$, or $\Gamma \vdash p : A \to B$ where $\Gamma = x_1 : \tau_1 \ldots x_n : \tau_n$ contains variable declarations.)

All functions and values are **total** (as opposed to **partial**), and **pure.**

E.g: **`div`** : $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ must handle every input.

The Curry-Howard-(Heyting-De Bruijn-Kolmogorov-…) correspondence

- Coq is based on **Type Theory** (Bertrand Russell, Per Martin-Löf, Thierry Coquand & Gérard Huet) a **unified** language where proofs & programs can be represented.

⇒ A **program** of **type** A → B is

a **term** p of **type** A → B

⇒ A **proof** of some **proposition** A ⇒ B : Prop is

a **term** p of **type** A → B

Formulae as Types (MLTT)

Logic (in Prop)	Proposition	Type	English	Example Term
Implication	$\Rightarrow$	$\to$	Function Space	`λ x : ℕ, x + x : ℕ → ℕ`
Universal Quantification	`∀ x : α, P`	`Π x : α, P`	Dependent Product	`λ x : ℕ, eq_refl x : Π x : ℕ, x = x`
Existential Quantification	`∃ x : α, P`	`Σ x : α, P`	Dependent Sum	`(0, eq_refl 0) : Σ x : ℕ, x = 0`
Truth	$\top$	`1, unit`	Unit Type	`() : 1`
Falsehood	$\bot$	`0, "void"`	Empty Type	`No term constructor`
Disjunction	`P ∨ Q`	`P + Q`	Sum Type	`(inl 0) : ℕ + 𝔹`
Conjunction	`P ∧ Q`	`P × Q`	Cartesian Product	`(0, `**`false`**`) : ℕ × 𝔹`

Type Constructors

Introduction :	Type	Elimination	Computation
`(λ x : α. b)`	$\alpha \rightarrow \beta$ / $\Pi x : \alpha, \beta$	`t : α ⊢` `f t : β[t/x]`	`(λ x : α. b) t` $\rightarrow_\beta$ `b[t/x]`
`(t, u)`	$\alpha_1 \times \alpha_2$ / $\Sigma x : \alpha_1, \alpha_2$	`p.1, p.2 :` α_i	$(x_1, x_2).i$ $\rightarrow_\iota$ x_i
`tt / I`	1 / True	`x : unit ⊢` `let tt := x in b`	`let tt := x in b` $\rightarrow_\iota$ `b`
`N/A`	0 / False	`x : False ⊢` `match x with end`	`N/A`

Booleans and Naturals

Type	Introduction	Elimination	Computation: Redex	Computation: Contractum

2 ($\mathbb{B}$)	`true`
	 `| false`
		`match x with`
		 `| true ⇒` t_0
		 `| false ⇒` t_1
		 `end`
			`match` c_j `with`
			 …
			 `|` c_i `=>` t_i
			 …
			 `end`
				$\rightarrow_\iota$ t_j

$\mathbb{N}$	`0 | S n`	`match x with`
		 `| 0 ⇒ t0`
		 `| S n ⇒ tS`
		 `end`
			`Example:`
			 `match S 0 with`
			 `| 0 ⇒ t0`
			 `| S n ⇒ tS`
			 `end`
				$\rightarrow_\iota$ `tS [0/n]`

The equality type

Introduction:

$$x : \alpha \vdash \mathtt{eq\_refl}\ x : \mathtt{eq}\ \alpha\ x\ x \qquad (\text{notation } x =_\alpha x)$$

Derivable:

- $x\ y : \alpha \vdash p : x = y \Rightarrow y = x$
- $x\ y\ z : \alpha \vdash p : x = y \Rightarrow y = z \Rightarrow x = z$
- $\vdash p : \forall\ f : \alpha \to \beta,\ \forall\ x\ y : \alpha,$
  $x = y \to f\ x = f\ y$

Equality is an equivalence relation and a congruence.

Summary

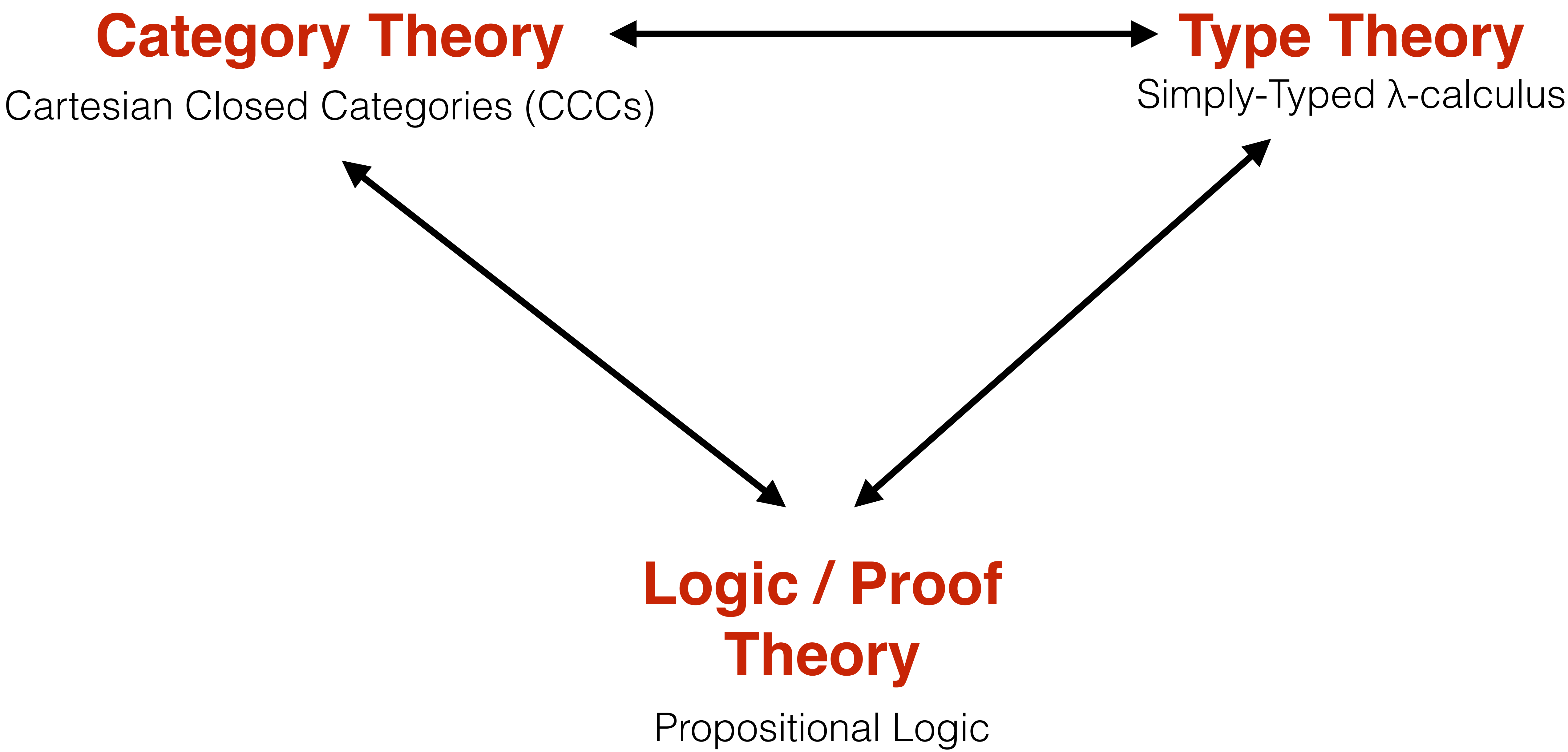
We have a **logic** with $\forall$, $\exists$, $\Rightarrow$, $\bot$, $\top$, $=$, and a **provability** relation $\vdash$.

An **algorithm** can check if $\Gamma \vdash t : T$ holds.

A **metatheoretical** result shows (relative) **consistency**: impossibility to construct a term p s.t. $\vdash p : \bot$ (i.e. without assuming extra axioms)

Type Theory gives a **unified** language in which we can express Higher-Order Logic **formulas** *and* construct machine-checked **proofs** for them.

The Trinity in the 70's



Trinity yesterday

Category Theory

LCCCs

Type Theory

Dependent Type Theory

Logic / Proof Theory

Intuitionistic Logic

Trinity these days

Higher Category Theory

Higher Topoï / ∞-groupoids

Homotopy Type Theory

(Voevodsky, Coquand, …)

Types as spaces, towards solving the gap with classical set theory

Logic / Proof Theory

Higher-Dimensional, Proof-Relevant Logic?

Type Theory with Inductive Types

In Coq, we have a general schema for defining datatypes, with the generic operators

`match .. with .. end` and `Fixpoint/fix`

We are going to see how to write proofs on them!

Inductive command

Inductive types generalize disjunction (sum types), conjunction (pairs), truth (unit) and falsehood (empty types).

For example, sums can be defined as:

```
Inductive sum (A B : Set) : Set :=
| inl : A → sum A B
| inr : B → sum A B.
```

Tactics

For any inductive type, we have the principles:

Constructors are disjoint: `discriminate`

Constructors are injective: `injection`

An induction principle: `induction`

Induction Principle

$$\forall\ (P : nat \to Prop)$$

$$(p0 : P\ 0)$$

$$(pS : \forall\ n,\ P\ n \to P\ (S\ n)),$$

$$\forall\ n,\ P\ n$$

```
λ (P : nat → Prop) p0 pS,
  fix prf n :=
    match n return P n with
    | 0 ⇒ (p0 : P 0)
    | S x ⇒ (pS x (prf x : P x)) : P (S x)
    end
```

Inductive Predicates

Inductive predicates allows to characterize a property of an object inductively:

```
Inductive even : nat → Prop :=
| even0 : even 0
| evenSS : forall n : nat, even n -> even (S (S n)).
```

Inversion

Inversion is the ability to infer which constructor/“rule” of the inductive predicate can apply to a particular situation.

Suppose you have `H : even (S (S k))`.

The only possible constructor to build such a value is `evenSS k H'` for some `H' : even k`

`inversion H` will destruct the hypothesis H to produce the possible subgoals for each applicable rule.

In case no constructor can apply, this solves the goal.

Inversion

Typical example:

```
Inductive lt : nat → nat → Prop :=
| lt0 : lt 0 (S n)
| ltS n m : lt n m -> lt (S n) (S m).
```

`inversion (H : lt n 0)` produces no subgoals.

Let’s switch to Coq

Going further: Dependently-Typed Programming

$$\text{div} : \forall\, (x : \mathbb{N})\ (y : \mathbb{N} \mid y \neq 0),$$

$$\{\ (q, r) \mid x = y * q + r \wedge r < y\ \}.$$

The function is total but requires a precondition on y.

$$\{\ x : \tau \mid P\ \} \equiv \Sigma\, x : \tau.\ P$$

I.e., we need to pass a pair of a value for y and a proof that it is non-zero.

We return not only the quotient and rest but also a **proof** that this really performs euclidian division.

Bibliography

Theory:

- Per Martin-Löf, Intuitionistic Type Theory (seminal)
- Programming in Martin-Löf Type Theory (Nordström, Petersson, and Smith, introductory, a classic)
- Proofs and Types (Girard, Lafont and Taylor, on the proof theory side)
- Categorical Logic (B. Jacobs, on semantics of type theories in categories)
- Semantics of Type Theory (T. Streicher, for the more set-theoretic expert)

Coq References:

- Software Foundations (Pierce et al, teaching material, CS-oriented, very accessible)
- Certified Programming with Dependent Types (Chlipala, MIT Press, DTP, Ltac automation)