An Introduction to Type Theory & Interactive Theorem Proving in Coq

coq.inria.fr

Matthieu Sozeau
Inria Pi.R2 & IRIF, Paris 7

EJCP 2017
27 June 2017, Toulouse

Get Coq Running

- In your browser: https://x80.org/collacoq (takes a while to load)
- Standalone IDE for Linux, Windows, Mac OS X: CoqIDE http://coq.inria.fr/download
- If you use opam: # opam install coq coqide
- Also an Emacs interface: ProofGeneral + company-coq

Slogan

Logic

~

Programming

Overview

1. **Theory**
   Dependent Type Theory and the Curry-Howard Isomorphism
   - History, Type vs Set theory
   - From simple to dependent type theory
   - Research on and applications of type theory
2. **Practice**
   Interactive Theorem Proving in Coq
   - Datatypes and propositions
   - Interactive proofs (with exercises)

Some History

- Beginning of 20th century: search for **formal foundations** of mathematics
- **Type Theory** descends from A. Church's simple theory of types, and Russell's type theory
- Alternative to Zermelo-Fraenkel's **Set Theory**

Logic & Computation

During the 20th century, variants of type theory are developed

- **Higher-Order Logic:** extends first-order logic of Set Theory with reasoning on predicates, relations, in a typed setting
- **Models of computation:** λ-calculi at the basis of functional programming languages.

A junction

Realization that **type systems** for programming languages are close to **logics**

- Simply typed λ-calculus correspond to propositional logic
- System F corresponds to second-order logic

Dependent Type Theory

- Per Martin-Löf introduced (intuitionistic) dependent type theory in the 70's.
- Strong departure from traditional Set Theory, in particular it is **constructive**.
- **Computation** at its core, mixing logic and computer science

$\Rightarrow$ Easier to mechanize:

Just another programming language!

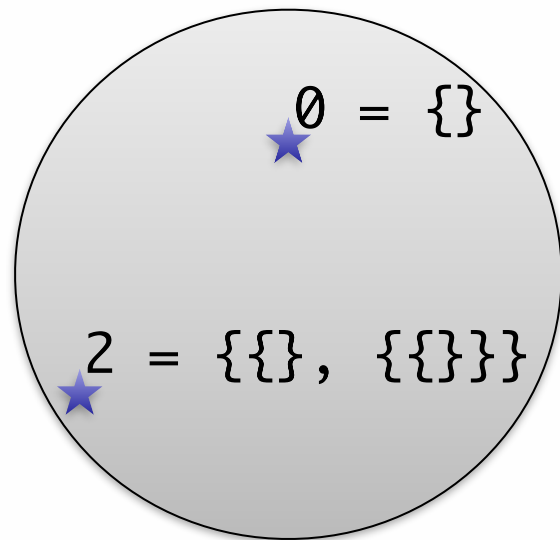
Type Theory

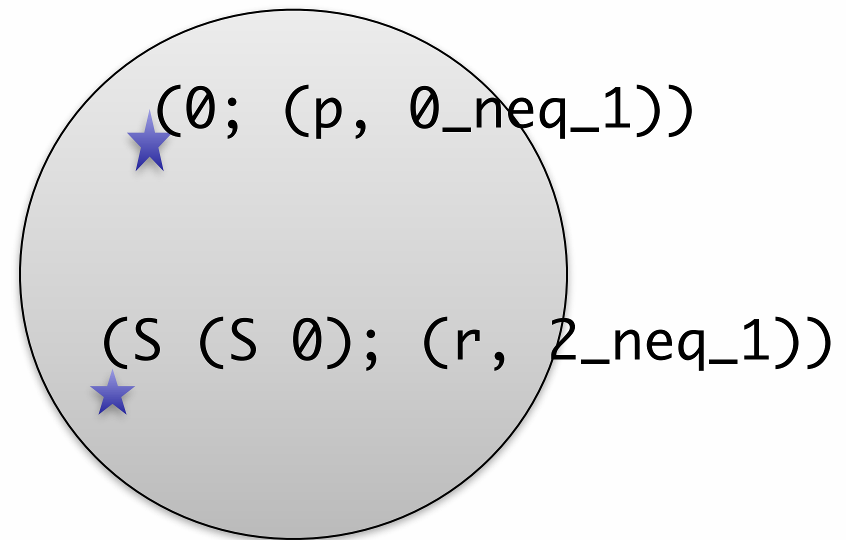
- Many refinements since the 70's, active area of research. Lots of applications and mature tools since the 90's.
- At the basis of proof assistants like Coq and Agda.
- Close cousin of HOL, NuPRL and PVS: different variants of type theory.

Type Theory Principles

- Types describe **structure** of their elements, $\neq$ elements of set theory
- Computation respects the types
- Pointcaré principle: $1 + 1 = 2$ is a mere **verification** of a result of computation. Proved by **definition** of addition.

Sets vs Types





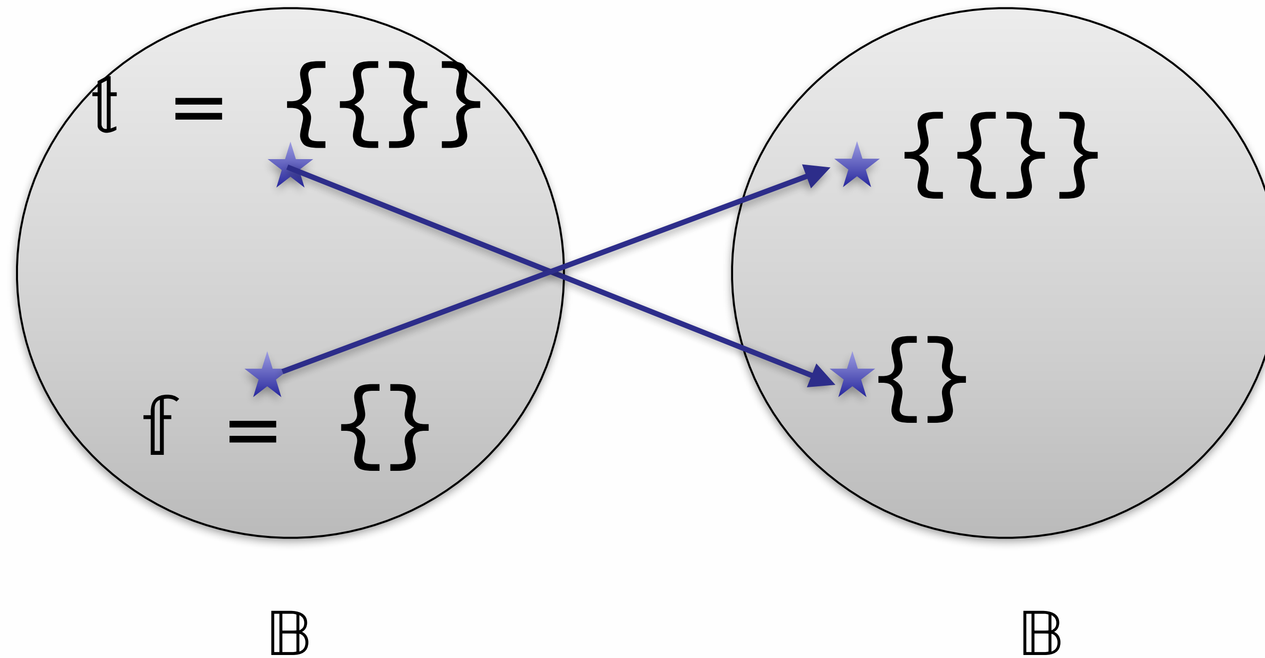
$\{ x \in \mathbb{N} : x <= 2 \wedge x \neq 1 \}$

$\{ x : \mathbb{N} \mid x <= 2 \times x \neq 1 \}$

- {}, {{}}, …
- $x \in \{ y \in R : y > 0 \}$
- $(x, y) = \{ \{x\}, \{x, y\} \} \in A \times B$
- $S \cup T = (S \cap T) \cup (S \setminus T) \cup (T \setminus S)$

- Inductive $\mathbb{N} := 0 : \mathbb{N} \mid S : \mathbb{N} \to \mathbb{N}$
- 0, (S 0), (S (S 0)), …
- $p : x > 0 \vdash (x; p) : \{ x : \mathbb{R} \mid y > 0\}$
- $x : A, y : B \vdash (x, y) : A \times B$

Sets vs Types



$\mathbb{B}$ $\mathbb{B}$

- neg = {{{}}, {{{}}, {}}}, {{}, {{}, {{}}}}} $\in \mathbb{B} \to \mathbb{B}$
- Theorem: Negation is involutive. For all booleans b, neg(neg(b)) = b.

- $\vdash$ **fun** b $\Rightarrow$ if b then 𝕗 else 𝕥 : $\mathbb{B} \to \mathbb{B}$
- $\vdash$ neginv : $\forall$ b : $\mathbb{B}$, neg (neg b) = b

Sets vs Types

Induction:

$P\ 0 \Rightarrow (\forall\ n, P\ n \Rightarrow P\ (S\ n)) \Rightarrow \forall\ n, P\ n$

Excluded middle: $\forall\ P, P \vee \neg\ P$

Existence principle:

$\exists\ x \in \mathbb{N}.\ P\ x \Rightarrow$ which x??

An **axiomatic** theory:

- Axioms in first-order logic and encodings
- No computation, only equational reasoning
- Proofs have no formal status.

Induction:

$\vdash \mathrm{Nind} : \forall\ P, P\ 0 \to (\forall\ n, P\ n \to P\ (S\ n)) \to \forall\ n, P\ n$

Excluded middle: $\nvdash \mathrm{em} : \forall\ P, P + \neg\ P$

Existence principle:

$\vdash t : \Sigma\ x : \mathbb{N}.\ P\ x \Rightarrow\ \vdash \mathrm{fst}\ t : \mathbb{N}.$

A **computational** theory:

- A computational language, **λ**-calculus
- Represents objects of interest **and** proofs about them in the same language.

FROM SIMPLE TO DEPENDENT TYPES

Our programming language

Terms of our λ-calculus have the grammar:

t ::= x (variable) | t u (application) | fun x : t ⇒ t (abstraction)

| Type | bool | true | false | if t then t else t | …

Computation substitution of "free" variables by terms: t[x:=u], e.g.:

(if x then false else true)[x:=true] ≡ if true then false else true

Our programming language

λ-calculus is:

- A **universal** computational language equivalent to Turing machines.
- Completely **pure**: no store or counters, just terms and a reduction/evaluation relation

Types and type systems for λ-calculus

We use a **type system** to verify the construction of **terms** in our programming language.

$\Gamma \vdash t : T$ expresses that **term** t has **type** T under **context** of variable declarations Γ (sometimes omitted). Types themselves have type "Type".

$$\vdash \text{bool} : \text{Type}$$

$$x : \text{bool} \vdash \text{if } x \text{ then false else true} : \text{bool}$$

Types and type systems for λ-calculus

Types you can depend on

- **Type safety**:
  If t has type T then any reduction of t has type T as well.
- **Normalization**
  Any program of type T reduces in a finite number of steps to a value of type T (e.g. true or false for a boolean)

Reading an inference rule

$$\frac{\Gamma \vdash t : T \ \ldots \quad \Gamma' \vdash u : U \ \ \text{(premisses)}}{\Gamma'' \vdash v : V \ \text{(conclusion)}}$$

$\leftrightarrow$

If t has type T under Γ, … and u has type U under Γ’,
then v has type V under Γ”

Unit type

$$\frac{}{\vdash \text{unit} : \text{Type}}$$

$$\frac{}{\vdash \text{tt} : \text{unit}}$$

⇒ unit has a single element, trivial type

$$\frac{\vdash x : \text{unit} \qquad \vdash y : T}{\vdash \textbf{match}\ x\ \textbf{with}\ \text{tt} \Rightarrow y\ \textbf{end} : T}$$

Reduction:

match tt **with** tt ⇒ y **end** → y

Function types

$$\frac{x : A \vdash b : B}{\vdash (\text{fun } x : A \Rightarrow b) : A \to B}$$

⇒ a **function** turning arguments of type A into terms of type B,

with function **application**

$$\frac{\vdash f : A \to B \quad \vdash a : A}{\vdash f\ a : B}$$

And **reduction** rule:

$(\text{fun } x \Rightarrow t)\ u \to_\beta t[x:=u]$

Function type example

$$\frac{x : \text{bool} \vdash x : \text{bool}}{\vdash (\text{fun } x : \text{bool} \Rightarrow x) : \text{bool} \rightarrow \text{bool}}$$

And **reduction**:

$$(\text{fun } x : \text{bool} \Rightarrow x)\ \text{true} \rightarrow_\beta x[x{:=}\text{true}] = \text{true}$$

Empty type

$$\frac{}{\vdash 0 : \text{Type}}$$

⇒ 0 has no values. We get a **function** turning any proof of 0 into an element of any type A:

$$A : \text{Type} \vdash \text{absurd} : 0 \to A$$

absurd = fun x : 0 ⇒ **match** x **with end**

- Equivalent of "assert false" in programming
  If we get an element of the empty type at some point in a program it means this point is actually unreachable.

Cartesian products

$$\frac{\vdash a : A \quad \vdash b : B}{\vdash (a, b) : A \times B}$$

⇒ a **pair** of a term of A and a term of B, coming with **projections**

$$\frac{\vdash p : A \times B}{\vdash \text{fst } p : A} \qquad \frac{\vdash p : A \times B}{\vdash \text{snd } p : B}$$

Reductions:

$$\text{fst } (a, b) \to a \qquad \text{snd } (a, b) \to b$$

Cartesian product example

$$\frac{\vdash 0 : \text{nat} \quad \vdash \text{true} : \text{bool}}{\vdash (0, \text{true}) : \text{nat} \times \text{bool}}$$

$\text{fst}\ (0, \text{true}) \to 0$ $\quad\quad$ $\text{snd}\ (0, \text{true}) \to \text{true}$

Sum types

$$\frac{\vdash a : A}{\vdash \text{Left } a : A + B} \qquad \frac{\vdash b : B}{\vdash \text{Right } b : A + B}$$

⇒ **either** a term of type A or a term of type B, coming with **case analysis:**

$$\frac{\vdash a : A + B \quad x : A \vdash c : C \quad y : B \vdash d : C}{\vdash \textbf{match } a \textbf{ with } \text{Left } x \Rightarrow c \mid \text{Right } y \Rightarrow d \textbf{ end} : C}$$

Reductions:

match Left a **with** Left x ⇒ c | Right y ⇒ d **end** → c[x:=a] …

Representing proofs

We want a language in which we can represent **proofs** as well as **objects of interest** (booleans, functions, pairs, etc..).

Idea: use the term structure to represent proof witnesses as well as programs.

A single language for both "universes"

The Brouwer-Heyting-Kolmogorov interpretation

Starting from intuitionistic propositional logic ($\top$, $\bot$, $\wedge$, $\vee$, $\Rightarrow$)

What is a proof of A $\wedge$ B?

$\Rightarrow$ a **pair** of a proof of A and a proof of B

What is a proof of A $\vee$ B?

$\Rightarrow$ **either** a proof of A or a proof of B

What is a proof of A $\Rightarrow$ B?

$\Rightarrow$ a **function** turning proofs of A into proofs of B.

The Brouwer-Heyting-Kolmogorov interpretation

The BHK interpretation shows that any **proof** of a **theorem** T of (intuitionistic) propositional logic can be translated to a **term** of **type** $[\![T]\!]$ in simply typed λ-calculus (the calculus with cartesian products, sum types and function types) by interpreting:

$$[\![\bot]\!] = 0 \qquad [\![\top]\!] = \text{unit}$$

$$[\![A \wedge B]\!] = [\![A]\!] \times [\![B]\!]$$

$$[\![A \vee B]\!] = [\![A]\!] + [\![B]\!]$$

$$[\![A \Rightarrow B]\!] = [\![A]\!] \to [\![B]\!]$$

Curry-Howard Isomorphism

The Proofs-as-Programs correspondence

Logic	λ-calculus
Proposition, Formula	Type T
Proof	Term M
M is a proof of P	$\vdash M : [\![P]\!]$
$A \wedge B$	$[\![A]\!] \times [\![B]\!]$

Computation and the disjunction property

A **constructive** logic:

Reducing a proof/term p such that

$\vdash p : A + B$

Will find either a Left a or Right b **value**, with a **witness** a for A or b for B.

Consistency

Logical Consistency of the system is ensured by proving that there is no closed term p such that

$$\vdash p : 0$$

I.e. the system is not contradictory.

Idea: if p existed then it could be reduced to a value of type 0, but there are none

Dependent Type Theory

DTT extends the **types** with dependencies on **terms** allowing to interpret equality of terms, existential quantification $\exists\, x : T.\, P$ and universal quantification $\forall\, x : T.\, P$.

	Logic	DTT
existential/ dependent sum	$\exists\, x : T.\ P$	$\Sigma\, x : T.\ [\![P]\!]$
universal/dependent product	$\forall\, x : T.\ P$	$\Pi\, x : T.\ [\![P]\!]$
equality	$x = y$	$x = y$

Equality Type

$$\frac{\vdash A : \mathrm{Type} \quad \vdash x : A \quad \vdash y : A}{\vdash x = y : \mathrm{Type}}$$

$$\frac{}{x : A \vdash \mathrm{eq_refl}\ x : x = x}$$

⇒ This type enjoys Leibniz's substitution principle:

$$P : A \to \mathrm{Type} \vdash \ldots : \forall\, x\, y : A, x = y \to P\, x \to P\, y$$

Equality Type examples

$$\frac{}{\vdash \text{eq_refl true} : \text{true} = \text{true}}$$

$$\frac{\vdash \ldots : x = y + 2 \qquad \vdash \text{eq_sym} : \forall\, x\, y, x + y = y + x}{\vdash \ldots\ (\text{eq_sym}\ y\ 2)\ \ldots : x = 2 + y}$$

$\Rightarrow$ Equational reasoning is available

Dependent Function types

$$\frac{x : A \vdash b : B}{\vdash \text{fun } x : A \Rightarrow b : \Pi x : A.\, B}$$

⇒ a **function** turning arguments of type A into terms of type B,

with function **application**

$$\frac{\vdash f : \Pi x : A.\, B \qquad \vdash a : A}{\vdash f\ a : B[x{:=}a]}$$

And **reduction** rule:

$$(\text{fun } x \Rightarrow t)\ u \to_\beta t[x{:=}u]$$

Dependent Function example

$$\frac{x : \text{bool} \vdash \text{eq_refl}\ x : x = x}{\vdash \text{fun}\ x : \text{bool} \Rightarrow \text{eq_refl}\ x : \Pi\, x : \text{bool}.\ x = x}$$

$$\vdash ((\text{fun}\ x : \text{bool} \Rightarrow \text{eq_refl}\ x)\ \text{true}) : (x = x)[x := \text{true}] \equiv \text{true} = \text{true}$$

⇒ We can now quantify on terms in our types, to talk about objects of interest in formulas.

Dependent Cartesian products

$$\frac{\vdash a : A \qquad \vdash b : B[x:=A]}{\vdash (a; b) : \Sigma x : A.\ B}$$

⇒ a **pair** of a term of A and a term of B, coming with **projections**

$$\frac{\vdash p : \Sigma x : A.\ B}{\vdash \mathrm{fst}\ p : A} \qquad \frac{\vdash p : \Sigma x : A.\ B}{\vdash \mathrm{snd}\ p : B[x:=\mathrm{fst}\ p]}$$

Reductions:

$\mathrm{fst}\ (a; b) \to a \qquad \mathrm{snd}\ (a; b) \to b$

Example term, type and proof

$\vdash$ fun x : nat $\Rightarrow$ if eqnat x O then true else false : nat $\to$ bool

$\vdash \Pi$ P Q : Type, ((P $\to$ Q) $\times$ P) $\to$ Q : Type

$\vdash$ (fun x : nat $\Rightarrow$ (x + 1, …)) : Π x : nat, Σ y : nat, y > x

Conversion

$$\frac{\vdash t : A \quad A \equiv B}{\vdash t : B}$$

$\Rightarrow$ $A \equiv B$ iff A and B are the same type **up-to reduction**

Embodies the **Pointcaré principle**:
some steps of reasoning are just computation not application of logical rules

Conversion example

$$\frac{\vdash \text{eq_refl}\ 2 : 2 = 2 \qquad (2 = 2) \equiv (1 + 1 = 2)}{\vdash \text{eq_refl}\ 2 : 1 + 1 = 2}$$

⇒ Conversion is **silent** in the term.

⇒ Can apply anywhere in a proof.

Summary

- Programming language with:
  $0, 1, \times, +, \rightarrow, \Pi, \Sigma$ and $=$
- Expresses a higher-order logic
- Types represent formulae/propositions
- Proofs represented as terms
- Normalization and consistency

FROM TYPE THEORY TO COQ

Propositions in Coq

Special universe of propositions "Prop", with specific connectives True, False, $\wedge$, $\vee$, $\exists$ (not 0, unit, $\times$, +, Σ, sorry for the confusion!)

$$\vdash \text{True}, \text{False} : \text{Prop}$$

$$\vdash \text{I} : \text{True}$$

$$A : \text{Type} \vdash \text{absurd} : \text{False} \to A$$

$$\vdash \vee, \wedge : \text{Prop} \to \text{Prop} \to \text{Prop}$$

$$\frac{x : A \vdash P : \text{Prop}}{\vdash \exists\, x : A.\, P : \text{Prop}}$$

Example terms, types and proofs

$\vdash (I, \text{fun } x : \text{True} \Rightarrow x) : \text{True} \wedge (\text{True} \rightarrow \text{True})$

$\vdash \Pi\ P\ Q : \text{Prop},\ ((P \rightarrow Q) \wedge P) \rightarrow Q : \text{Prop}$

$\vdash \text{ltS} : \Pi\ x : \text{nat}, x < x + 1$

$\vdash (\text{fun } x : \text{nat} \Rightarrow \text{exist } (x + 1)\ (\text{ltS } x)) :$

$\Pi\ x : \text{nat}, \text{exists } y : \text{nat}, x < y$

Propositions in Coq

- Propositions are "**proof-irrelevant**" (consistent axiom)

  $$\forall\ (A : Prop)\ (p\ q : A),\ p = q$$

  Exact shape of proof terms doesn't matter: only existence.

- Prop is **impredicative**:

  $\vdash idP : \forall A : Prop,\ A \to A : Prop$

  $\vdash idP\ (\forall A : Prop,\ A \to A)\ idP : \forall A : Prop,\ A \to A$

  Logical strength (but somewhat mysterious semantically)

- Propositions are **erasable** by **extraction**

The type theory of Coq

- General **recursive datatypes** (Inductive and Coinductive types) and **predicates/relations**.
  e.g. enumerations, numbers, lists, arbitrary tree-like structures, streams, inductive relations, orders
- Not just one "Type" category
  An infinite hierarchy of Type(i) for i $\in \mathbb{N}$, used to prevent paradoxes.
  The system lets the user be ambiguous and use "Type" for every one of those.
- Record types, universe polymorphism, an ML-like module system and type classes.

Inductive datatypes

General tree-like data structures
$\leftrightarrow$ OCaml and Haskell's algebraic data types:

- Inductive lists:

  Inductive list (A : Type) : Type :=
  | nil : list A
  | cons (a : A) (l : list A) : list A

  $\vdash$ cons 1 (cons 2 nil) : list nat

- Fixpoint construction to program recursive definitions on them.

Inductive families

General schema to define n-ary relations.

- Example: inductive definition of ≤ order on natural numbers.

  Inductive le : nat → nat → Prop :=
   | leO : forall x : nat, le x x
   | leS : forall x y : nat, le x y → le x (S y)

  "Smallest" relation generated by these rules.

- Used to define inductive **properties** of objects

DTT as a programming language

- **Total:** no non-terminating, non-productive or partial functions.
  Running a program/proof cannot diverge or raise a runtime exception (in theory)
- **Pure:** no side-effects like mutable data-structures, explicit memory management or IO (like Haskell, requires monadic programming).
  Running a program always returns the same value (like any mathematical function).

Dependently Typed Programming

Terms in types allow specifying **invariants** in the types of programs, e.g:

Pre-post conditions:

$$\vdash \text{euclid} : \forall\, x\ y : \text{nat},\ y > 0 \to \{\ (q, r) \mid x = y * q + r\ \}$$

Static bounds checking:

$$\vdash \text{array_get} : \forall\ \text{len}\ (a : \text{array bool len})\ i,\ i < \text{len} \to \text{bool}$$

Extraction to ML/Haskell removes the logical content:

$$\text{val euclid} : \text{nat} \to \text{nat} \to \text{nat} * \text{nat}$$

Proof-checking
=
Type-checking

- Type-checking and type inference are **decidable** in Coq, hence proof checking is too.
- Proving becomes a program refinement game:

$$\vdash\ ?: \Pi\ n\ a\ b\ c : nat,\ n > 2 \rightarrow a \neq 0 \rightarrow b \neq 0 \rightarrow c \neq 0 \rightarrow a\verb|^|n + b\verb|^|n = c\verb|^|n \rightarrow False$$

DTT for proof assistants

- Obvious **difficulty**: proof terms are hard to write correctly, just like programs. Tend to be large and full of details.
- **tactics** are used to produce the proof terms **automatically** and **incrementally**.
  meta-programs producing terms in the type theory. See part II.

Coq tactics

- High-level tactics to decide particular theories (congruence closure, first-order logic, ring equations…)
- Various **tactic languages** to express proofs
- Unification and proof-search engines to produce proofs

RESEARCH AND APPLICATIONS OF TYPE THEORY

Research in type theory

- **Models** of various (dependent) type theories (metamathematical studies). No one "true" dependent type theory.
- Links with computational **effects**, e.g. memory, reactive, concurrent programming, randomness.
- **Homotopy Type Theory** (V. Voevodsky et al, The Univalent Foundations Program)
  Hot topic of research for re-founding mathematics on a type theory basis. Set Theory as a subset of Type Theory. Cubical Type Theory (T. Coquand et al)
- Links with Computational **Complexity** theory.

DTT for Mechanized Mathematics

Rich language & logic for mathematical structures, statements about them and proofs. Notable developments:

- Proofs of the 4-color theorem (G.Gonthier, B.Werner - Inria - Microsoft Research), and the Feit-Thompson theorem (G.Gonthier et al.- Inria - Microsoft Research).
  Large part of university level algebra formalized.
- CCoRN algebra and analysis library (Nijmegen)
- Kepler's conjecture about "how to stack oranges optimally" in Isabelle/HOL (10 years project).

DTT for Computer Science

Two trends: **verification** of research results (e.g. type safety proofs, or that particularly tricky lemma) and **construction** of certified software (compilers, analysers, trace checkers, OS kernels…)

- Libraries about automata, circuits, rewriting, probabilistic computation, …: models of computation
- Frameworks for verification of programs and specifications: from assembly (Bedrock, VeLLVM MIT) to C (VST, IRIS) to JavaScript (JSCert)

Correct-by-construction software

- Formalisation of semantics of JavaCard , certification of security functionalities (Gemalto, Trusted Labs)
- Certified C compiler producing optimized code (Compcert, X. Leroy et al, Inria). Compiler produced by **extraction**
- sel4, first certified micro-kernel implementation. CertiKOS: certified OS kernel, full-stack system verification, from scheduling to file-systems.
- Development of certified static analysers (D.Pichardie et al - Inria)

Ressources for learning Coq

- https://coq.inria.fr/documentation
- **Coq'Art** (Castéran & Bertot, Inria)
- **Certified Programming with Dependent Types** (Chlipala, MIT)
- **Software Foundations** course (3 volumes, from basic proofs to verification of complex algorithms), easy to use to learn alone. Used in graduate courses in the USA.

Related Tools

- **Agda**: a dependently-typed programming language
  More focused on programming than mathematical proofs, proof-term interface only, includes experimental type theory extensions
- **Idris**: another dependently-typed programming language
  Meant as a Haskell-with-dependent types, focused on treating side-effects like IO
- **Isabelle/HOL**: another prominent proof assistant based on Higher-Order Logic, using a classical foundation
- **Matita**, **Lean**: interactive theorem provers based on similar theories to Coq.

Conclusion

- A unified formal language for structures, programs, specifications and proofs.
- Mix logic and computation
- General purpose, from program verification to mathematical proofs.
- Next up: how it works in practice

Get Coq Running

- In your browser: https://x80.org/collacoq
- Standalone IDE for Linux, Windows, Mac OS X: CoqIDE http://coq.inria.fr/download
- If you use opam: # opam install coq coqide
- Also an Emacs interface: ProofGeneral + company-coq
- https://www.irif.fr/~sozeau/EJCP17_demo.v