Introduction to Dependent Type Theory and Higher-Order Logic

Gilles Barthe

INRIA Sophia-Antipolis, France

Dependent type theory

A language for:

- defining mathematical objects (including algorithms)
- performing computations on and with these objects
- reasoning about these objects

It is a foundational language that underlies:

- several proof assistants (inc. Coq, Epigram, Agda)
- a few programming languages (inc. Cayenne, DML).

Objectives

- to introduce the formalism of dependent type theory;
- to present convertibility relation from the point of view of the user and of the proof-assistant implementor;
- to describe methods for defining and checking recursive functions.

Proof assistants

- Implement type theories/higher order logics to specify and reason about mathematics.
- Feature expressive specification languages that allow encoding of complex data structures and mathematical theories.
- Interactive proofs, with mechanisms to guarantee that
  - theorems are applied with the right hypotheses
  - functions are applied to the right arguments
  - no missing cases in proofs or in function definitions
  - no illicit logical step (all reasoning is reduced to elementary steps)

  Proof assistants include domain-specific tactics that help solving specific problems efficiently.
- Completed proofs are represented by proof objects that can easily be checked by a small, trusted proof-checker. Such proof assistants provide the highest correctness guarantees.

Sample applications

- Programming:
  - JavaCard platform, including JCVM and BCV
  - C compiler
  - Program verification for Java and C programs
- Cryptographic protocols
  - Dolev-Yao model (perfect cryptography assumption)
  - Generic Model and Random Oracle Model
- Mathematics and logic:
  - Galois theory, category theory, real numbers, polynomials, computer algebra systems, geometry, etc.
  - 4-colors theorem
  - Type theory

Type theory and the Curry-Howard isomorphism

- Type theory is a programming language in which to write algorithms.
- All functions are total and terminating, so that convertibility is decidable.
- Type theory is a language for proofs, via the Curry-Howard isomorphism:

$$\begin{array}{ccc} \text{Propositions} & = & \text{Types} \\ \text{Proofs} & = & \text{Terms} \\ \text{Proof-Checking} & = & \text{Type-Checking} \end{array}$$

- The underlying logic is constructive. However, classical logic can be recovered with an axiom, or better with a control operator (see e.g. Griffin POPL'90, Murthy Ph.D.)

A Theory of Functions: Syntax and Judgments

- Types: $\begin{array}{rcll} \mathcal{T} & = & \mathcal{B} & \text{(Base type)} \\ & | & \mathcal{T} \to \mathcal{T} & \text{(Function Type)} \end{array}$
- Expressions: $\mathcal{E} = \mathcal{V} \mid \mathcal{E}\ \mathcal{E} \mid \lambda\mathcal{V} : \mathcal{T}.\ \mathcal{E}$
- Judgements:

$$\overbrace{x_1 : A_1, \quad \ldots \quad , x_n : A_n}^{\textit{Context}} \vdash \overbrace{M}^{\textit{Subject}} : \overbrace{B}^{\textit{Predicate}}$$

M is a function of type B provided $x_i : A_i$ for $i = 1 \ldots n$

A Theory of Functions: Typing Rules

$$\frac{}{\Gamma \vdash x : A} \quad (x : A) \in \Gamma$$

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

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

A (Core) Programming Language

- The result of applying a function to an argument may be computed, using β-reduction

$$(\lambda x : A.\ M)\ N \quad \rightarrow_\beta \quad M\{x := N\}$$

- The result of computing an algorithm is unique, because β-reduction is confluent

$$M =_\beta N \quad \Rightarrow \quad M \downarrow_\beta N$$

Properties of the Type System

- Subject Reduction:

$$\Gamma \vdash M : A \wedge M \to_\beta N \quad \Rightarrow \quad \Gamma \vdash N : A$$

- Strong Normalization: if $\Gamma \vdash M : A$ then there is no infinite sequence of β-reduction steps starting from M

$$M \to_\beta M_1 \to_\beta M_2 \to_\beta M_3 \ldots$$

Properties of the Type System (ctd)

- Convertibility: if $\Gamma \vdash M, N : A$ then it is decidable whether $M =_\beta N$.
- Type-Checking: it is decidable whether $\Gamma \vdash M : A$.
- Type-Inference: partial function inf from contexts and terms to types, and such that for all $A \in \mathcal{T}$

$$\Gamma \vdash M : A$$
$$\Leftrightarrow$$
$$\Gamma \vdash M : (\mathrm{inf}(\Gamma, M)) \wedge (\mathrm{inf}(\Gamma, M)) = A$$

Minimal Intuitionistic Logic

- Formulae: $$\begin{array}{lcll} \mathcal{F} & = & \mathcal{X} & \text{(propositional variable)} \\ & \mid & \mathcal{F} \to \mathcal{F} & \text{(implication)} \end{array}$$
- Judgements $A_1, \ \ldots \ , A_n \vdash B$
- Derivation rules

$$\frac{}{\Gamma \vdash A} \quad A \in \Gamma$$

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

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

BHK Interpretation

A proof of:	is given by:
$A \wedge B$	a proof of A and a proof of B
$A \vee B$	a proof of A or a proof of B
$A \rightarrow B$	a method to transform proofs of A into proofs of B
$\forall x.\ A$	a method to produce a proof of $A(t)$ for every t
$\perp$	has no proof

A Language for Proofs

- If $\Gamma \vdash M : A$ then $\Gamma \vdash A$
- If $\Gamma \vdash A$ then $\Gamma \vdash M : A$ for some M
- A one-to-one correspondence between derivation trees and λ-terms (next slide)
- Proof normalisation $\Leftrightarrow$ β-reduction
- In a proof assistant M is often built backwards.

The correspondence between proofs and λ-terms

$$A \qquad\qquad \frac{}{\Gamma \vdash x : A} \quad (x : A) \in \Gamma$$

$$\frac{\begin{array}{c}\vdots\, d \\ A \to B\end{array} \qquad \begin{array}{c}\vdots\, d' \\ A\end{array}}{B} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

$$\frac{\begin{array}{c}[A]^i \\ \vdots\, d \\ B\end{array}}{A \to B}\, i \qquad\qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\, M : A \to B}$$

Also correspondence between proof normalization and β-reduction

Conjunction

- Types $\mathcal{T} := \ \dots \mid \mathcal{T} \times \mathcal{T}$
- Expressions $\mathcal{E} := \ \dots \ \mid \pi_1\ \mathcal{E} \mid \pi_2 \mathcal{E} \mid \ \langle \mathcal{E}, \mathcal{E} \rangle$
- Reduction rules $\pi_i \langle M_1, M_2 \rangle \quad \to_\pi \quad M_i$
- Typing rules

$$\frac{\Gamma \ \vdash \ M_1 : A_1 \quad \Gamma \ \vdash \ M_2 : A_2}{\Gamma \ \vdash \ \langle M_1, M_2 \rangle : A_1 \times A_2}$$

$$\frac{\Gamma \ \vdash \ M : A_1 \times A_2}{\Gamma \ \vdash \ \pi_i\ M : A_i}$$

Disjunction

- Types $\mathcal{T} := \ \ldots \ \mid \mathcal{T} + \mathcal{T}$
- Expressions

$$\mathcal{E} := \ \ldots \mid \iota_1\ \mathcal{E} \mid \iota_2\ \mathcal{E} \mid (\text{case } \mathcal{E} \text{ of } (\iota_1 \mathcal{V}) \Rightarrow \mathcal{E} \mid (\iota_2 \mathcal{V}) \Rightarrow \mathcal{E} \text{ end})$$

- Typing rules $\dfrac{\Gamma \vdash M : A_i}{\Gamma \vdash \iota_i\ M : A_1 + A_2}$

$$\frac{\Gamma \vdash M : A_1 + A_2 \quad \Gamma, x_i : A_i \vdash P_i : B}{\Gamma \vdash \text{case } M \text{ of } (\iota_1 x_1) \Rightarrow P_1 \mid (\iota_2 x_2) \Rightarrow P_2 \text{ end} : B}$$

- Reduction rule

  $\text{case } (\iota_i\ M) \text{ of } (\iota_1 x_1) \Rightarrow P_1 \mid (\iota_2 x_2) \Rightarrow P_2 \text{ end} \rightarrow_\iota P_i\{x_i := M\}$

Universal quantification

- Let A be a set. Then a predicate B over A is a function that associates to every element of a a type, namely the type of proofs of $B\ a$.
- Let A be a set and let B be a predicate over A. Then a proof of $\forall a : A.\ B\ a$ is a function that associates to every $a : A$ a proof, i.e. an inhabitant, of $B\ a$ (Brouwer-Heyting-Kolmogorov interpretation of proofs).
- Universal quantification corresponds to generalized function space, like implication corresponds to function space.

Rule for generalized function space and universal quantification

- First approximation:

$$\frac{\Gamma, a : A \vdash M : \ B}{\Gamma \vdash \lambda a{:}A.\ M : \Pi a{:}A.\ B}$$

$$\frac{\Gamma \vdash M : \Pi a{:}A.\ B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ \ N : B\{a := N\}}$$

- Correct version: in the abstraction rule, check that the product type is well-formed

Calculus of Constructions as typed higher-order logic

Many variants of higher-order logics, reflecting different choices, e.g.:

- Should sets and propositions be separated?
- Should the universe of propositions be a set?
- Should we quantify over functions, predicates, both?

Here: the Calculus of Constructions.

- Distinguishes between sets and propositions

$$\mathcal{S} = \{\mathbf{Prop}, \mathbf{Type}\}$$

- The universe of propositions is a set

$$\mathcal{A} = \{(\mathbf{Prop} : \mathbf{Type})\}$$

- The formalization of mathematics requires using a hierarchy of universes $\mathbf{Type}_i$ instead of $\mathbf{Type}$.

Terms and reduction

- Expressions

$$\mathcal{E} := \mathcal{V} \mid \mathcal{S} \mid \lambda\mathcal{V}{:}\mathcal{E}.\,\mathcal{E} \mid \mathcal{E}\ \mathcal{E} \mid \Pi\mathcal{V}{:}\mathcal{E}.\,\mathcal{E}$$

  (implication $A \rightarrow B$ is an abbreviation for $\Pi x{:}A.\,B$ when x does not occur free in B)

- β-reduction

$$(\lambda x{:}A.\,M)\ N \qquad \rightarrow_\beta \qquad M\{x := N\}$$

- β-equality $=_\beta$ is the reflexive, symmetric, transitive closure of $\rightarrow_\beta$

Typing rules

$$\langle\rangle \vdash s_1 : s_2 \qquad (s_1, s_2) \in \mathcal{A}$$

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \quad x \notin \Gamma$$

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} \quad x \notin \Gamma$$

Typing rules (ctd)

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x{:}A.\ B) : s_3} \qquad (s_1, s_2, s_3) \in \mathcal{R}$$

$$\frac{\Gamma \vdash F : (\Pi x{:}A.\ B) \quad \Gamma \vdash a : A}{\Gamma \vdash F\ a : B\{x := a\}}$$

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

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s}{\Gamma \vdash A : B'} \qquad B =_\beta B'$$

Remarks

- Contexts are *ordered* lists: $A : \mathbf{Type}, x : A \vdash x : A$ is valid but $x : A, A : \mathbf{Type} \vdash x : A$ is not.
- Types are not defined a priori, and furthermore they have a computational behavior.
- The (conv) rule ensures that convertible types have the same inhabitants, and allows to replace reasoning by computation.
- Rules for Calculus of Constructions:
  - Function space: $(\mathbf{Type}, \mathbf{Type})$,
  - Implication: $(\mathbf{Prop}, \mathbf{Prop})$,
  - Quantification: $(\mathbf{Type}, \mathbf{Prop})$
  - $(\mathbf{Prop}, \mathbf{Type})$

Polymorphism

- Many proofs are parametric: $\lambda x{:}\alpha.\, x : \alpha \to \alpha$
- The $(\mathbf{Type}, \mathbf{Prop})$-rule allows to parameterize λ-terms over types: $\lambda\alpha{:}\mathbf{Prop}.\, \lambda x{:}\alpha.\, x$
- Impredicative encoding of connectives and datastructures (see next slide)
- If we add the rule $(\mathbf{Type}_1, \mathbf{Type}_0)$ in order to parameterize functions over types, we obtain an inconsistent system known as U^-: in this system $\perp$ is inhabited. Yet the rule below is safe

$$\frac{\Gamma \vdash A : \mathrm{Type}_1 \quad \Gamma, x : A \vdash B : \mathbf{Type}_0}{\Gamma \vdash (\Pi x{:}A.\, B) : \mathbf{Type}_1}$$

Impredicative encoding of connectives/free structures

$$
\begin{array}{rcl}
\top & \equiv & \Pi\alpha : \mathbf{Prop}.\alpha \to \alpha \\
\bot & \equiv & \Pi\alpha : \mathbf{Prop}.\alpha \\
\wedge & \equiv & \lambda A, B{:}\mathbf{Prop}.\ \Pi\alpha : \mathbf{Prop}.A \to B \to \alpha \\
\vee & \equiv & \lambda A, B{:}\mathbf{Prop}.\ \Pi\alpha : \mathbf{Prop}.(A \to \alpha) \to (B \to \alpha) \to \alpha \\
\neg & \equiv & \lambda A{:}\mathbf{Prop}.\ A \to \bot \\
\Rightarrow & \equiv & \lambda A, B : \mathbf{Prop}{:}A \to B. \\
\mathsf{Nat} & \equiv & \Pi\alpha{:}\mathbf{Prop}.\ \alpha \to (\alpha \to \alpha) \to \alpha \\
0 & \equiv & \lambda\alpha{:}\mathbf{Prop}.\ \lambda x{:}\alpha.\ \lambda f{:}\alpha \to \alpha.\ x \\
\mathsf{succ} & \equiv & \lambda n{:}\mathsf{Nat}.\ \lambda\alpha{:}\mathbf{Prop}.\ \lambda x{:}\alpha.\ \lambda f{:}\alpha \to \alpha.\ f\ (n\ \alpha\ x\ f)
\end{array}
$$

Types depending on terms

- Terms may occur in types, e.g.:

$$\begin{array}{ll} & N : \mathbf{Type}, O : \mathbf{N}, P : N \to \mathbf{Prop} \\ \vdash & \lambda x : (P\ O).\ x : (P\ O) \to P((\lambda z : \mathbf{N}.\ z)\ O) \end{array}$$

- Useful for dependently typed data structures, e.g.

$$\begin{array}{ll} & A : \mathbf{Type}, N : \mathbf{Type}, n : N, Vec : N \to \mathbf{Type} \to \mathbf{Type} \\ \vdash & Vec\ n\ A : \mathbf{Type} \end{array}$$

- Existential quantification: for $T : \mathbf{Type}$, we have

$$\begin{array}{lll} \exists_T & \equiv & \lambda P{:}T \to \mathbf{Prop}. \\ & & \quad : (\Pi\alpha : \mathbf{Prop}.\Pi x : T.((P\ x) \to \alpha) \to \alpha) \end{array}$$

Inductive Definitions

- Impredicative encoding of free structures is unsatisfactory
- Mechanisms to define data structures and principles to define recursive functions and reason by induction
- Recursive functions must be terminating

Approaches to inductive definitions

- Recursors
- Case-expressions and guarded fixpoints
- Pattern-matching

All share the same basic rules. In the case of natural numbers:

$$\vdash \text{Nat} : s \qquad \vdash 0 : \text{Nat} \qquad \frac{\Gamma \vdash n : \text{Nat}}{\Gamma \vdash S\ n : \text{Nat}}$$

Recursors: typing and reduction rules

$$\frac{\Gamma \vdash f_0 : A \qquad \Gamma \vdash f_s : \mathrm{Nat} \to A \to A}{\Gamma \vdash \mathrm{RecN}(f_0, f_s) : \mathrm{Nat} \to A}$$

$$\frac{\begin{array}{c} \Gamma \vdash P : \mathrm{Nat} \to s \\ \Gamma \vdash f_0 : P\ 0 \qquad \Gamma \vdash f_s : \Pi x{:}\mathrm{Nat}.\ (P\ n) \to P(S\ n) \end{array}}{\Gamma \vdash \mathrm{RecN}(f_0, f_s) : \Pi x{:}\mathrm{Nat}.\ P\ n}$$

$$\begin{array}{rcl} \mathrm{RecN}(f_0, f_s)\ \ 0 & \to_\iota & f_0 \\ \mathrm{RecN}(f_0, f_s)\ \ (S\ x) & \to_\iota & f_s\ \ x\ \ (\mathrm{RecN}(f_0, f_s)\ \ x) \end{array}$$

Case expressions and fixpoints: typing rules

$$\frac{\Gamma \vdash n : \mathrm{Nat} \quad \Gamma \vdash f_0 : A \quad \Gamma \vdash f_s : \mathrm{Nat} \to A}{\Gamma \vdash \mathrm{case}\ n\ \mathrm{of}\{0 \Rightarrow f_0 \mid s \Rightarrow f_s\} : A}$$

$$\frac{\begin{array}{c}\Gamma \vdash n : \mathrm{Nat} \quad \Gamma \vdash P : \mathrm{Nat} \to s \\ \Gamma \vdash f_0 : P\ 0 \quad \Gamma \vdash f_s : \Pi n{:}\mathrm{Nat}.\ P\ (S\ n)\end{array}}{\Gamma \vdash \mathrm{case}\ n\ \mathrm{of}\{0 \Rightarrow f_0 \mid s \Rightarrow f_s\} : P\ n}$$

$$\frac{\Gamma, f : \mathrm{Nat} \to A \vdash e : \mathrm{Nat} \to A}{\Gamma \vdash \mathrm{letrec}\ f = e : \mathrm{Nat} \to A}$$

Case expressions and fixpoints: reduction rules

$$\begin{array}{rcl}
\text{case } 0 \text{ of} \{0 \Rightarrow e_0 \mid s \Rightarrow e_s\} & \rightarrow & e_0 \\
\text{case } (s \;\; n) \text{ of} \{0 \Rightarrow e_0 \mid s \Rightarrow e_s\} & \rightarrow & e_s \;\; n \\
(\text{letrec } f = e) \;\; n & \rightarrow & e\{f := (\text{letrec } f = e)\} \;\; n
\end{array}$$

Fixpoints may not terminate. To recover termination, we must

- use a side condition $\mathcal{G}(f, e)$, read f is guarded in e, in the typing rule for fixpoint
- require n to be of the form $c \; \vec{b}$ in the reduction rule in the reduction rule for fixpoint

Pattern matching

Define a function by writing down equations:

$$\begin{array}{rcl} & plus : \mathrm{Nat} \rightarrow \mathrm{Nat} \rightarrow \mathrm{Nat} & \\ plus\ O\ y & \longrightarrow & y \\ plus\ (S\ x)\ y & \longrightarrow & S\ (plus\ x\ y) \end{array}$$

It is a very convenient way to define functions. Under some hypothesis, may be reduced to case-expressions and fixpoints.

Beyond natural numbers

Inductive definitions encode a rich class of structures:

- algebraic types: booleans, binary natural numbers, integers, etc
- parameterized types: lists, trees, etc
- inductive families and relations: vectors, accessibility relations (to define functions by well-founded recursion)

Parameterized types

- Lists
  Inductive List (A:**Type**):=
  nil: List A
  | cons: A $\rightarrow$ List A$\rightarrow$ List A
- Trees (not accepted verbatim in CIC)
  Inductive List (A:**Type**):=
  empty: Tree A
  | branch List (Tree A) $\rightarrow$ Tree A

Higher-order types: ordinals

Inductive Ord:=
O: Ord
| S : Ord $\to$ Ord
| lim: (Nat $\to$ Ord) $\to$ Ord

Inductive families: vectors

Inductive Vec (n:Nat; A:**Type**):=
empty: Vec 0 A
| cons: Π m:Nat. A $\rightarrow$ Vec n A$\rightarrow$ Vec (n+1) A

Inductive families: AVL trees

Trees where the difference between the sons of a node is ≤ 1

Inductive BT (n:Nat):=
empty: BT 0
| branch: Π m,n:Nat. max(m,n)-min(m,n)$\leq$ 1 $\rightarrow$
BT n$\rightarrow$ BT m$\rightarrow$ BT (max(m,n)+1)

Inductive types not allowed in dependent type theory

- Types with non-positive constructors

  Inductive Terms:=

  var: Nat $\rightarrow$ Terms

  | app: Terms $\rightarrow$ Terms $\rightarrow$ Terms

  | lambda: (Terms $\rightarrow$ Terms) $\rightarrow$ Terms

- Nested datatypes

  Inductive Nest (A:**Type**):=

  nil: Nest A

  | cons: A $\rightarrow$ Nest (A $\times$ A)$\rightarrow$ Nest A

Structural vs. well-founded recursion

Many recursive definitions are not structurally recursive, e.g.

letrec qs $= \lambda$ l:(List Nat).
case l of { nil $\Rightarrow$ nil
| (cons a l) $\Rightarrow$ (qs (filter ($<$ a) l))$++$ a $++$ (qs (filter ($\geq$ a) l)) }

How to encode quicksort?

- As a relation? No.
- As a function using a general accessibility predicate
- As a function using a specific predicate.

Using general accessibility

- General accessibility predicate:
  Inductive Acc (R:A $\rightarrow$ A $\rightarrow$ **Prop**): A $\rightarrow$ **Prop**:=
  acc_intro : Π a:A. (Π b:B. R b a $\rightarrow$ Acc R b) $\rightarrow$ Acc R a
- Take R to be the relation such that R (filter ($<$ a) l) a::l and R (filter ($\geq$ a) l) a::l, and define using elimination for Acc a "partial" function
  qs_aux:Π l:(List Nat). Acc R l $\rightarrow$ (List Nat)
- Show that the function is "total", i.e.
  Π l:(List Nat). Acc R l

Using a specific accessibility predicate

- General accessibility predicate:
  Inductive Acc_qs: (List Nat $\rightarrow$ List Nat $\rightarrow$ **Prop**:=
  | acc_qs1: Π l:List Nat. Π a:Nat. (filter ($<$ a) l) a::l
  | acc_qs2: Π l:List Nat. Π a:Nat. (filter ($\geq$ a) l) a::l
- Define using elimination for Acc_qs a "partial" function qs_aux:Π l:(List Nat). Acc_qs l $\rightarrow$ (list Nat)
- Show that the function is "total", i.e. Π l:(List Nat). Acc_qs l

Nested functions

How to encode functions such as nest

$$\begin{array}{rcl} \mathsf{nest}\ 0 & \longrightarrow & 0 \\ \mathsf{nest}\ (\mathsf{S}\ \mathsf{n}) & \longrightarrow & \mathsf{nest}\ (\mathsf{nest}\ \mathsf{n}) \end{array}$$

Problem: we must define the function at the same time as its domain.
Solution: use inductive-recursive definitions.

Reasoning about recursive functions

Inductively defined relations come with induction and inversion principles. What about functions?

- For general recursive functions, we can prove statements by induction and inversion on the ad-hoc accessibility predicate
- For structurally recursive functions, we can generate induction and inversion principles automatically (e.g. functional induction in Coq).
- Difficulties (Coq specific): case analysis over patterns is reduced to case analysis over constructors, and default cases are expanded.

More on inductive types

- Proof theoretical explanations
- Formal syntax and rules:
  - positivity conditions
  - elimination schemes
- Infinite objects in type theory, e.g. streams.

Strong sums

- Σ-types are dependent products: an element of $\Sigma x : A.\ B$ is a pair $a : A$ and $b : B\{x := A\}$
- Σ-types are used to encode subsets and mathematical theories

Syntax for strong sums

- Expressions

$$T := \ \ldots \mid \Sigma x : T.\ T \mid \langle T, T \rangle \mid \pi_1\ T \mid \pi_2\ T$$

- Reduction

$$\begin{array}{lcl} \pi_1 \langle t_1, t_2 \rangle & \to & t_1 \\ \pi_2 \langle t_1, t_2 \rangle & \to & t_2 \end{array}$$

Typing rules

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Sigma x : A.B : s_3} \quad (s_1, s_2, s_3) \in \mathcal{R}'$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : A \qquad \Gamma \vdash t_2 : B\{x := t_1\} \\ \Gamma \vdash \Sigma x : A.B : s\end{array}}{\Gamma \vdash \langle t_1, t_2 \rangle : \Sigma x : A.B}$$

$$\frac{\Gamma \vdash t : \Sigma x : A.B}{\Gamma \vdash \pi_1\ t : A}$$

$$\frac{\Gamma \vdash t : \Sigma x : A.B}{\Gamma \vdash \pi_2\ t : B\{x := \pi_1\ t\}}$$

Theories

$$\begin{aligned} \mathit{Setoid} &\equiv \Sigma T : \mathbf{Type}.\Sigma R : T \to T \to \mathbf{Prop}.\mathrm{eqrel}(R) \\ \mathit{Monoid} &\equiv \Sigma A : \mathit{Setoid}.\Sigma o : \mathrm{bmap}\ A.\Sigma e : \mathrm{el}\ A.\phi \end{aligned}$$

Note: structures can be encoded as inductive types with one constructor

Meta-theoretical properties

- Confluence
- Subject reduction
- Strong normalization
- Consistency
- Decidability of convertibility and type-checking

Calculus of Inductive Constructions: enough or too much?

- In general, it is not enough: universes are needed for formalizing mathematical structures, equality is often too weak, no easy representation of quotients or subsets, difficult to change between representations, etc.
- For many problems, it is too much: we do not need dependent types, complex inductive definitions, etc.
- Yet it is useful to fall back on a powerful theory when needed: some seemingly basic facts about programs may require deep mathematical results, which may be conveniently expressed in type theory