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 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:

- Propositions $= \text{Types}$
- Proofs $= \text{Terms}$
- Proof-Checking $= \text{Type-Checking}$

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.)
Types: $\mathcal{T} = B$ (Base type)  
| $\mathcal{T} \rightarrow \mathcal{T}$ (Function Type)  

Expressions: $\mathcal{E} = V | \mathcal{E} \mathcal{E} | \lambda V : \mathcal{T}. \mathcal{E}$  

Judgements:  

\[
\begin{array}{c}
\textbf{Context} \\
\underbrace{x_1 : A_1, \ldots, x_n : A_n} \\
\textbf{Subject} \\
\underbrace{M} \\
\textbf{Predicate} \\
\underbrace{B}
\end{array}
\]

$M$ is a function of type $B$ provided $x_i : A_i$ for $i = 1 \ldots n$
A Theory of Functions: Typing Rules

\[ \Gamma \vdash x : A \quad (x : A) \in \Gamma \]

\[ \Gamma \vdash M : A \to B \quad \Gamma \vdash N : A \]
\[ \Gamma \vdash M \, N : B \]

\[ \Gamma, x : A \vdash M : B \]
\[ \Gamma \vdash \lambda x : A. \, M : A \to B \]
The result of applying a function to an argument may be computed, using $\beta$-reduction

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

The result of computing an algorithm is unique, because $\beta$-reduction is confluent

$$M =_\beta N \ \Rightarrow \ M \downarrow_\beta N$$
Properties of the Type System

- **Subject Reduction**: 
  \[ \Gamma \vdash M : A \land M \rightarrow_{\beta} N \quad \Rightarrow \quad \Gamma \vdash N : A \]

- **Strong Normalization**: if \( \Gamma \vdash M : A \) then there is no infinite sequence of \( \beta \)-reduction steps starting from \( M \)
  \[ M \rightarrow_{\beta} M_1 \rightarrow_{\beta} M_2 \rightarrow_{\beta} M_3 \ldots \]
- 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 $A \in \mathcal{T}$

$$\Gamma \vdash M : A \iff \Gamma \vdash M : (\inf(\Gamma, M)) \land (\inf(\Gamma, M)) = A$$
Minimal Intuitionistic Logic

- Formulae: \( \mathcal{F} = \chi \) (propositional variable)
  
  \[ \mathcal{F} \to \mathcal{F} \] (implication)

- Judgements \( A_1, \ldots, A_n \vdash B \)

- Derivation rules

\[
\begin{align*}
\Gamma \vdash A & \quad A \in \Gamma \\
\Gamma \vdash A \to B & \quad \Gamma \vdash A \\
\Gamma \vdash B \\
\Gamma, A \vdash B \\
\Gamma \vdash A \to B
\end{align*}
\]
### BHK Interpretation

<table>
<thead>
<tr>
<th>A proof of:</th>
<th>is given by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \land B$</td>
<td>a proof of $A$ and a proof of $B$</td>
</tr>
<tr>
<td>$A \lor B$</td>
<td>a proof of $A$ or a proof of $B$</td>
</tr>
<tr>
<td>$A \rightarrow B$</td>
<td>a method to transform proofs of $A$ into proofs of $B$</td>
</tr>
<tr>
<td>$\forall x. A$</td>
<td>a method to produce a proof of $A(t)$ for every $t$</td>
</tr>
<tr>
<td>$\bot$</td>
<td>has no proof</td>
</tr>
</tbody>
</table>
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 $\lambda$-terms (next slide)

Proof normalisation $\Leftrightarrow$ $\beta$-reduction

In a proof assistant $M$ is often built backwards.
The correspondence between proofs and $\lambda$-terms

$$\Gamma \vdash x : A \quad (x : A) \in \Gamma$$

\[
\begin{array}{c}
A \\
\vdash d : A \rightarrow B \\
\vdash A \\
\hline
\vdash B \\
\end{array}
\]
\[
\begin{array}{c}
A \rightarrow B \\
\vdash d' : A \\
\vdash \Gamma \vdash M : A \rightarrow B \\
\vdash \Gamma \vdash N : A \\
\hline
\vdash \Gamma \vdash M \cdot N : B \\
\end{array}
\]

Also correspondence between proof normalization and $\beta$-reduction

$$\begin{array}{c}
[A]^i \\
\vdash d : B \\
\vdash B \\
\hline
\vdash A \rightarrow B^i \\
\end{array}$$

$$\begin{array}{c}
\Gamma, x : A \vdash M : B \\
\hline
\vdash \Gamma \vdash \lambda x : A.\ M : A \rightarrow B \\
\end{array}$$
Conjunction

- Types $\mathcal{T} ::= \ldots | \mathcal{T} \times \mathcal{T}$
- Expressions $\mathcal{E} ::= \ldots | \pi_1 \mathcal{E} | \pi_2 \mathcal{E} | \langle \mathcal{E}, \mathcal{E} \rangle$
- Reduction rules $\pi_i \langle M_1, M_2 \rangle \rightarrow_\pi M_i$
- Typing rules

\[
\begin{align*}
\Gamma \vdash M_1 : A_1 & \quad \Gamma \vdash M_2 : A_2 \\
\Gamma \vdash \langle M_1, M_2 \rangle : A_1 \times A_2 \\
\Gamma \vdash M : A_1 \times A_2 & \\
\Gamma \vdash \pi_i M : A_i
\end{align*}
\]
Disjunction

- Types $\mathcal{T} := \ldots | \mathcal{T} + \mathcal{T}$
- Expressions
  \[
  \mathcal{E} := \ldots \mid \nu_1 \mathcal{E} \mid \nu_2 \mathcal{E} \mid (\text{case } \mathcal{E} \text{ of } (\nu_1 \nu) \Rightarrow \mathcal{E} \mid (\nu_2 \nu) \Rightarrow \mathcal{E} \text{ end})
  \]
- Typing rules
  \[
  \frac{\Gamma \vdash M : A_i}{\Gamma \vdash \nu_i M : A_1 + A_2}
  \]
  \[
  \frac{\Gamma \vdash M : A_1 + A_2 \quad \Gamma, x_i : A_i \vdash P_i : B}{\Gamma \vdash \text{case } M \text{ of } (\nu_1 x_1) \Rightarrow P_1 \mid (\nu_2 x_2) \Rightarrow P_2 \text{ end} : B}
  \]
- Reduction rule
  \[
  \text{case } (\nu_i M) \text{ of } (\nu_1 x_1) \Rightarrow P_1 \mid (\nu_2 x_2) \Rightarrow P_2 \text{ end} \rightarrow \nu_i P_i\{x_i := M\}
  \]
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:

\[
\Gamma, a : A \vdash M : B \\
\Gamma \vdash \lambda a : A. M : \Pi a : A. B
\]

\[
\Gamma \vdash M : \Pi a : A. B \quad \Gamma \vdash N : A \\
\Gamma \vdash M \ N : B\{a := N\}
\]

- Correct version: in the abstraction rule, check that the product type is well-formed
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
  
  \[ S = \{\text{Prop}, \text{Type}\} \]

- The universe of propositions is a set
  
  \[ A = \{(\text{Prop} : \text{Type})\} \]

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

Gilles Barthe
Dependent Type Theory and Higher-Order Logic
Expressions

\[ E ::= V \mid S \mid \lambda V. E \mid E \ E \mid \Pi V. E. E \]

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

\( \beta \)-reduction

\[
(\lambda x:A. M) \ N \rightarrow_\beta M\{x := N\}
\]

\( \beta \)-equality \( =_\beta \) is the reflexive, symmetric, transitive closure of \( \rightarrow_\beta \)
Typing rules

\[
\begin{align*}
& \langle \rangle \vdash s_1 : s_2 & (s_1, s_2) \in A \\
& \frac{}{\Gamma \vdash A : s} & x \notin \Gamma \\
& \frac{}{\Gamma, x : A \vdash x : A} & x \notin \Gamma \\
& \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} & x \notin \Gamma
\end{align*}
\]
Typing rules (ctd)

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

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

\[
\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A. B) : s \\
\Gamma \vdash \lambda x : A. b : \Pi x : A. B
\]

\[
\Gamma \vdash A : B \quad \Gamma \vdash B' : s \\
\Gamma \vdash A : B' \quad B =_{\beta} B'
\]
Contexts are ordered lists: $A : \textbf{Type}$, $x : A \vdash x : A$ is valid but $x : A$, $A : \textbf{Type} \vdash x : A$ 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: $(\textbf{Type}, \textbf{Type})$,
- Implication: $(\textbf{Prop}, \textbf{Prop})$,
- Quantification: $(\textbf{Type}, \textbf{Prop})$
- $(\textbf{Prop}, \textbf{Type})$
Many proofs are parametric: $\lambda x:\alpha. x : \alpha \to \alpha$

The $(\text{Type}, \text{Prop})$-rule allows to parameterize $\lambda$-terms over types: $\lambda \alpha: \text{Prop}. \lambda x: \alpha. x$

Impredicative encoding of connectives and datastructures (see next slide)

If we add the rule $(\text{Type}_1, \text{Type}_0)$ in order to parameterize functions over types, we obtain an inconsistent system known as $U^-$: in this system $\bot$ is inhabited. Yet the rule below is safe

$$
\Gamma \vdash A : \text{Type}_1 \quad \Gamma, x : A \vdash B : \text{Type}_0 \\
\Gamma \vdash (\Pi x : A. B) : \text{Type}_1
$$
Impredicative encoding of connectives/free structures

\[ \top \equiv \Pi \alpha : \text{Prop}. \alpha \to \alpha \]
\[ \bot \equiv \Pi \alpha : \text{Prop}. \alpha \]
\[ \land \equiv \lambda A, B : \text{Prop}. \Pi \alpha : \text{Prop}. A \to B \to \alpha \]
\[ \lor \equiv \lambda A, B : \text{Prop}. \Pi \alpha : \text{Prop}. (A \to \alpha) \to (B \to \alpha) \to \alpha \]
\[ \neg \equiv \lambda A : \text{Prop}. A \to \bot \]
\[ \Rightarrow \equiv \lambda A, B : \text{Prop}. A \to B. \]
\[ \text{Nat} \equiv \Pi \alpha : \text{Prop}. \alpha \to (\alpha \to \alpha) \to \alpha \]
\[ 0 \equiv \lambda \alpha : \text{Prop}. \lambda x : \alpha. \lambda f : \alpha \to \alpha. x \]
\[ \text{succ} \equiv \lambda n : \text{Nat}. \lambda \alpha : \text{Prop}. \lambda x : \alpha. \lambda f : \alpha \to \alpha. f (n \alpha x f) \]
Types depending on terms

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

\[ N : \text{Type}, \ O : \ N, \ P : N \rightarrow \text{Prop} \]
\[ \vdash \ \lambda x : (P \ O). \ x : (P \ O) \rightarrow P((\lambda z : N. \ z) \ O) \]

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

\[ A : \text{Type}, \ N : \text{Type}, \ n : N, \ Vec : N \rightarrow \text{Type} \rightarrow \text{Type} \]
\[ \vdash \ \text{Vec} \ n \ A : \text{Type} \]

- Existential quantification: for \( T : \text{Type} \), we have

\[ \exists T \equiv \lambda P: T \rightarrow \text{Prop}. \]
\[ : (\Pi \alpha : \text{Prop}. \Pi x : T.((P \ x) \rightarrow \alpha) \rightarrow \alpha) \]
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:

\[
\begin{align*}
\Gamma & \vdash \text{Nat} : s \\
\Gamma & \vdash 0 : \text{Nat} \\
\Gamma, n : \text{Nat} & \vdash S\, n : \text{Nat}
\end{align*}
\]
Recursors: typing and reduction rules

\[
\Gamma \vdash f_0 : A \quad \Gamma \vdash f_s : \text{Nat} \to A \to A
\]

\[
\Gamma \vdash \text{RecN}(f_0, f_s) : \text{Nat} \to A
\]

\[
\Gamma \vdash P : \text{Nat} \to \text{Nat} \\
\Gamma \vdash f_0 : P \ 0 \\
\Gamma \vdash f_s : \Pi x: \text{Nat}. (P \ n) \to P(S \ n)
\]

\[
\Gamma \vdash \text{RecN}(f_0, f_s) : \Pi x: \text{Nat}. P \ n
\]

\[
\text{RecN}(f_0, f_s) \ 0 \ \to_\ell \ f_0 \\
\text{RecN}(f_0, f_s) \ (S \ x) \ \to_\ell \ f_s \ x \ (\text{RecN}(f_0, f_s) \ x)
\]
Case expressions and fixpoints: typing rules

\[ \Gamma \vdash n : \text{Nat} \quad \Gamma \vdash f_0 : A \quad \Gamma \vdash f_s : \text{Nat} \rightarrow A \]
\[ \Gamma \vdash \text{case } n \text{ of } \{ 0 \Rightarrow f_0 \mid s \Rightarrow f_s \} : A \]

\[ \Gamma \vdash n : \text{Nat} \quad \Gamma \vdash P : \text{Nat} \rightarrow s \]
\[ \Gamma \vdash f_0 : P \ 0 \quad \Gamma \vdash f_s : \Pi n: \text{Nat}. \ P (S \ n) \]
\[ \Gamma \vdash \text{case } n \text{ of } \{ 0 \Rightarrow f_0 \mid s \Rightarrow f_s \} : P \ n \]

\[ \Gamma, f : \text{Nat} \rightarrow A \vdash e : \text{Nat} \rightarrow A \]
\[ \Gamma \vdash \text{letrec } f = e : \text{Nat} \rightarrow A \]
Case expressions and fixpoints: reduction rules

\[
\begin{align*}
\text{case 0 of } & \{0 \Rightarrow e_0 \mid s \Rightarrow e_s\} & \rightarrow & e_0 \\
\text{case } (s \, n) \text{ of } & \{0 \Rightarrow e_0 \mid s \Rightarrow e_s\} & \rightarrow & e_s \, n \\
\text{(letrec } f = e) \, n & \rightarrow & e\{f := \text{letrec } f = e\} \, n
\end{align*}
\]

Fixpoints may not terminate. To recover termination, we must

- use a side condition $G(f, e)$, 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 for fixpoint
Define a function by writing down equations:

\[\text{plus} : \text{Nat} \to \text{Nat} \to \text{Nat}\]

\[
\begin{align*}
\text{plus } O \ y & \rightarrow \ y \\
\text{plus } (S \ x) \ y & \rightarrow \ S \ (\text{plus } x \ y)
\end{align*}
\]

It is a very convenient way to define functions. Under some hypothesis, may be reduced to case-expressions and fixpoints.
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
  \[
  \text{Inductive} \ \text{List} \ (A: \textbf{Type}) :=
  \]
  \[
  \text{nil: List A} \\
  \mid \text{cons: A \rightarrow List A \rightarrow List A}
  \]

- Trees (not accepted verbatim in CIC)
  \[
  \text{Inductive} \ \text{List} \ (A: \textbf{Type}) :=
  \]
  \[
  \text{empty: Tree A} \\
  \mid \text{branch List (Tree A) \rightarrow Tree A}
  \]
Inductive Ord :=
O : Ord
| S : Ord → Ord
| lim : (Nat → Ord) → Ord
Inductive families: vectors

**Inductive** Vec (n: Nat; A: Type):=
empty: Vec 0 A
| cons: Π m: Nat. A → Vec n A → Vec (n+1) A
Trees where the difference between the sons of a node is $\leq 1$

**Inductive** $\text{BT} \ (n:\text{Nat}):=$

- $\text{empty}: \ 	ext{BT} \ 0$
- $\text{branch}: \ \Pi \ m,n:\text{Nat}. \ \max(m,n)-\min(m,n) \leq 1 \rightarrow$
  $\text{BT} \ n \rightarrow \text{BT} \ m \rightarrow \text{BT} \ (\max(m,n)+1)$
- Types with non-positive constructors
  \textbf{Inductive} Terms :=
  \begin{align*}
  \text{var: Nat} & \rightarrow \text{Terms} \\
  \text{app: Terms} & \rightarrow \text{Terms} \rightarrow \text{Terms} \\
  \text{lambda: (Terms} & \rightarrow \text{Terms)} \rightarrow \text{Terms}
  \end{align*}

- Nested datatypes
  \textbf{Inductive} Nest (A: Type) :=
  \begin{align*}
  \text{nil: Nest A} \\
  \text{cons: A} & \rightarrow \text{Nest (A} \times A) \rightarrow \text{Nest A}
  \end{align*}
Many recursive definitions are not structurally recursive, e.g.

\[
\text{letrec } qs = \lambda l: (\text{List Nat}). \\
\text{case } l \text{ of } \begin{cases} 
\text{nil } \Rightarrow \text{nil} \\
\text{cons } a \ l \Rightarrow (qs (\text{filter } (< a) \ l)) ++ a ++ (qs (\text{filter } (\geq a) \ l)) 
\end{cases}
\]
How to encode quicksort?

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

**Inductive** \( \text{Acc} \) \((R:A \rightarrow A \rightarrow \text{Prop})\): \( A \rightarrow \text{Prop} \):

\[
\text{acc_intro} : \prod a:A. (\prod b:B. R b a \rightarrow \text{Acc} R b) \rightarrow \text{Acc} R a
\]

Take \( R \) to be the relation such that \( R \) \((\text{filter} (< a) l) a::l \) and \( R \) \((\text{filter} (\geq a) l) a::l \), and define using elimination for \( \text{Acc} \) a

“partial” function

\( qs_{aux}:\prod l:(\text{List Nat}). \text{Acc} R l \rightarrow (\text{List Nat}) \)

Show that the function is “total”, i.e.

\( \prod l:(\text{List Nat}). \text{Acc} R l \)
General accessibility predicate:

**Inductive** Acc\_qs: (List Nat \to List Nat \to Prop\) :=

| acc\_qs1: \( \Pi \text{l:List Nat. } \Pi \text{a:Nat.} \ (\text{filter} \ (< \text{a}) \ \text{l}) \ \text{a::l} \)

| acc\_qs2: \( \Pi \text{l:List Nat. } \Pi \text{a:Nat.} \ (\text{filter} \ (\geq \text{a}) \ \text{l}) \ \text{a::l} \)

- Define using elimination for Acc\_qs a “partial” function
  qs\_aux: \( \Pi \text{l:(List Nat). } \text{Acc\_qs l} \rightarrow (\text{list Nat}) \)

- Show that the function is “total”, i.e.
  \( \Pi \text{l:(List Nat). } \text{Acc\_qs l} \)
How to encode functions such as `nest`

\[
\begin{align*}
\text{nest } 0 & \rightarrow 0 \\
\text{nest } (S\ n) & \rightarrow \text{nest } (\text{nest } n)
\end{align*}
\]

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

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

### Expressions

\[ T ::= \ldots | \Sigma x : T. T | \langle T, T \rangle | \pi_1 T | \pi_2 T \]

### Reduction

\[
\begin{align*}
\pi_1 \langle t_1, t_2 \rangle & \rightarrow t_1 \\
\pi_2 \langle t_1, t_2 \rangle & \rightarrow t_2
\end{align*}
\]
Typing rules

\[
\begin{align*}
\Gamma \vdash A : s_1 & \quad \Gamma, x : A \vdash B : s_2 \\
\Gamma \vdash \Sigma x : A. B : s_3 & \quad (s_1, s_2, s_3) \in \mathcal{R}'
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash t_1 : A & \quad \Gamma \vdash t_2 : B\{x := t_1\} \\
\Gamma \vdash \Sigma x : A. B : s & \quad \Gamma \vdash \langle t_1, t_2 \rangle : \Sigma x : A. B
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash t : \Sigma x : A. B & \quad \Gamma \vdash \pi_1 \ t : A
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash t : \Sigma x : A. B & \quad \Gamma \vdash \pi_2 \ t : B\{x := \pi_1 \ t\}
\end{align*}
\]
\begin{align*}
\text{Setoid} & \equiv \Sigma T : \text{Type}. \Sigma R : T \to T \to \text{Prop}. \text{eqrel}(R) \\
\text{Monoid} & \equiv \Sigma A : \text{Setoid}. \Sigma o : \text{bmap } A. \Sigma e : \text{el } A. \phi
\end{align*}

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
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.