Advanced (co)induction in dependent type theory modulo rewriting

Frédéric Blanqui (INRIA)

Place. Deducteam | LSV | ENS Paris-Saclay Currently in Cachan, the ENS and the LSV will move in April 2020 to Gif-sur-Yvette, rue Noetzlin, near PCRI, IUT Orsay and Centrale Supélec.

Context. Lambdapi is a new proof assistant based on a logical framework called the \( \lambda \Pi \)-calculus modulo rewriting, which is an extension of the simply-typed \( \lambda \)-calculus (the basis of functional programming languages like OCaml or Haskell) with dependent types (e.g. vectors and matrices of some given dimension) and an equivalence relation on types generated by user-defined rewrite rules \[3\]. Thanks to rewriting, Lambdapi allows the formalization of proofs that cannot be done in other proof assistants (e.g. simplicial sets of infinite dimensions).

However, there is currently no user support for doing inductive proofs. Currently, the user must define induction principles by hand. This can quickly become heavy when there are many constructors, and difficult when one considers complex inductive types.

To take a simple example, from the following definition:

\begin{verbatim}
inductive Nat : TYPE = // natural numbers
| 0 : Nat
| s : Nat => Nat
\end{verbatim}

we would like the following code to be internally generated:

\begin{verbatim}
constant symbol Nat : TYPE
constant symbol 0 : Nat
constant symbol s : Nat => Nat
symbol ind_Nat : \forall p, \pi(p0) => (\forall n, \pi(pn) => \pi(p(sn))) => \forall n, \pi(pn)
rule ind_Nat p p0 ps 0 => p0
and ind_Nat p p0 ps (s n) => ps n (ind_Nat p p0 ps n)
\end{verbatim}

where \( p : \text{Nat} \Rightarrow \text{Prop} \) is a predicate on \( \text{Nat} \) (\( \text{Prop} \) is the type of propositions) and \( \pi : \text{Prop} \Rightarrow \text{TYPE} \) maps propositions to types.

Goal. The goal of the internship is to automate the definition of the induction principle associated to an inductive type \[15\].

As a follow-up, one could consider how to handle co-inductive types \[3\] and co-recursion \[1\] in Dedukti as well, that is, terms representing infinite objects like streams.
One could also develop tactics for proving some inductive theorems automatically using a technique based on rewriting \([7, 14]\). This technique requires sufficient completeness of function definitions \([17]\) a property that is useful also for proving the consistency of a logical system \([6]\).

**Workplan.** One can start by considering the case of a simple first-order inductive type like \(\text{Nat}\), and then consider increasingly more complex classes of inductive types:

- **polymorphic inductive types**

  \[
  \text{inductive} \quad \text{List}: \text{Set} \Rightarrow \text{TYPE} \equiv \begin{cases} \text{// lists} \\ \text{nil}(\alpha): \text{List} \ \alpha \\ \text{cons}(\alpha): \tau \ \alpha \Rightarrow \text{List} \ \alpha \Rightarrow \text{List} \ \alpha \end{cases}
  \]

  where \(\text{Set}\) is the type of sets and \(\tau: \text{Set} \Rightarrow \text{TYPE}\) maps sets to types.

- **dependent inductive types**

  \[
  \text{inductive} \quad V: \text{Set} \Rightarrow \text{Nat} \Rightarrow \text{TYPE} \equiv \begin{cases} \text{// vectors} \\ \text{nil}(\alpha): \text{Vec} \ \alpha \ 0 \\ \text{cons}(\alpha): \tau \ \alpha \Rightarrow \forall n, \text{Vec} \ \alpha \ n \Rightarrow \text{Vec} \ \alpha \ (s \ n) \end{cases}
  \]

- **mutually defined inductive types**

  \[
  \text{inductive} \quad \text{Tree}: \text{TYPE} \equiv \begin{cases} \text{// trees with finite branching} \\ \text{leaf}: \text{Tree} \\ \text{node}: \text{Forest} \Rightarrow \text{Tree} \\ \text{and} \quad \text{Forest}: \text{TYPE} \equiv \begin{cases} \text{//} \\ \text{empty}: \text{Forest} \\ \text{cons}: \text{Tree} \Rightarrow \text{Forest} \Rightarrow \text{Forest} \end{cases} \end{cases}
  \]

- **strictly positive inductive types**

  \[
  \text{inductive} \quad \text{Ord}: \text{TYPE} \equiv \begin{cases} \text{// ordinals} \\ \text{0}: \text{Ord} \\ \text{s}: \text{Ord} \Rightarrow \text{Ord} \\ \sup: (\text{Nat} \Rightarrow \text{Ord}) \Rightarrow \text{Ord} \end{cases}
  \]

- **positive inductive types** \([4]\)

  \[
  \text{inductive} \quad \text{Rou}: \text{TYPE} \equiv \begin{cases} \text{// continuations} \\ \text{over}: \text{Rou} \\ \text{next}: ((\text{Rou} \Rightarrow \text{List} \ \text{nat}) \Rightarrow \text{List} \ \text{nat}) \Rightarrow \text{Rou} \end{cases}
  \]

- **nested inductive types** \([13]\)

  \[
  \text{inductive} \quad \text{Bush}: \text{Set} \Rightarrow \text{TYPE} \equiv \begin{cases} \text{//} \\ \text{nil}(\alpha): \text{Bush} \ \alpha \\ \text{cons}(\alpha): \tau \ \alpha \Rightarrow \text{Bush} \ (\text{Bush} \ \alpha) \Rightarrow \text{Bush} \ \alpha \end{cases}
  \]
• inductive predicates

\[
\text{inductive } \leq : \text{Nat} \Rightarrow \text{Nat} \Rightarrow \text{Prop} := \\
| 0 \leq : \forall x, \pi(0 \leq x) \\
| s \leq : \forall x, y, \pi(x \leq y) \Rightarrow \pi(s \ x \leq s \ y)
\]

• inductive-inductive types [11]

\[
\text{inductive } \text{SortL} : \text{TYPE} := \ // \ sorted \ lists \\
| \text{nil} : \text{SortL} \\
| \text{cons} : \forall y \ 1 \ (p: \text{Le} \ y \ 1), \text{SortL} \\
\text{and} \ \text{Le} : \text{Nat} \Rightarrow \text{SortL} \Rightarrow \text{TYPE} := \\
| \text{Le-nil} : \forall x, \text{Le} \ x \ \text{nil} \\
| \text{Le-cons} : \forall x, y \ 1 \ (p: \text{Le} \ y \ 1), \\
\ x \leq y \Rightarrow \text{Le} \ x \ 1 \Rightarrow \text{Le} \ x \ (\text{cons} \ y \ 1 \ p)
\]

• inductive-recursive definitions [8]

\[
\text{inductive } \text{UniqL} : \text{TYPE} := \ // \ lists \ with \ unique \ elements \\
| \text{nil} : \text{UniqL} \\
| \text{cons} : \forall x \ 1, \ x \neq 1 \Rightarrow \text{UniqL} \\
\text{and} \ \neq : \text{Nat} \Rightarrow \text{UniqL} \Rightarrow \text{TYPE} := \\
\text{modulo} \ x \neq \text{nil} \Rightarrow \text{True} \\
\ x \neq \text{cons} \ y \ 1 \_ \Rightarrow \ x \neq y \land x \neq 1
\]

• the combination of both [9]

• quotient types [10]

\[
\text{inductive } \text{Bag} : \text{TYPE} := \ // \ multisets \\
| \text{nil} : \text{Bag} \\
| \text{cons} : \text{Nat} \Rightarrow \text{Bag} \Rightarrow \text{Bag} \\
\text{modulo} := \\
| \text{swap} : \forall x, y, b, \text{cons} \ x \ y \ b = \text{cons} \ y \ x \ b
\]

where \( = \) is Leibniz’s polymorphic equality (\(x=y\) iff \(x\) and \(y\) satisfies the same propositions).

• higher dimensional inductive types [16]

\[
\text{inductive } \text{S2} : \text{TYPE} := \ // \ homotopic \ sphere \\
| \text{base} : \text{S2} \\
\text{modulo} := \\
| \text{surf} : \text{eq} (\text{eq} \ \text{base} \ \text{base}) \ \text{refl} \ \text{refl}
\]

where \( \text{eq} \) is Leibniz equality and \( \text{refl} \) the canonical reflexivity proof.

• cyclic data types [12]

\[
\text{inductive } \text{CList} : \text{TYPE} := \ // \ cyclic \ lists \\
| \text{nil} : \text{CList} \\
| \text{cons} : \text{Nat} \Rightarrow \text{CList} \Rightarrow \text{CList} \\
\text{modulo} := \ // \ the \ axioms \ of \ cyclicity
\]
• co-inductive types [1]

```plaintext
coinductive Stream : TYPE // streams of natural numbers
| head : Stream ⇒ N
| tail : Stream ⇒ Stream

countant symbol zeros : Stream // stream of 0
head zeros ⇒ 0
tail zeros ⇒ zeros
```

An interesting working example is to define in Lambdapi the set of well-typed terms of \( \lambda \Pi \) itself [2], the simplest dependent type theory, and compare it with its definition in Agda.

**Requirements.** Some familiarity with a functional programming language with pattern-matching.

**References**


[15] E. Tassi. Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq. [https://hal.inria.fr/hal-01897468](https://hal.inria.fr/hal-01897468), 2019.
