Adding support for induction in Dedukti

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 λΠ-calculus modulo rewriting, which is an extension of the simply-typed λ-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:

\[
\text{inductive } N : \text{TYPE} ::= \begin{array}{l}
\text{// natural numbers} \\
\mid 0 : N \\
\mid s : N \Rightarrow N
\end{array}
\]

we would like the following code to be internally generated:

\[
\begin{array}{l}
\text{constant symbol } N : \text{TYPE} \\
\text{constant symbol } 0 : N \\
\text{constant symbol } s : N \Rightarrow N \\
\text{symbol ind}_N : \forall p, \pi (p0) \Rightarrow (\forall n, \pi (pn) \Rightarrow \pi (p(sn))) \Rightarrow \forall n, \pi (pn) \\
\text{rule ind}_N p p0 ps 0 \rightarrow p0 \\
\text{and ind}_N p p0 ps (s n) \rightarrow ps n \ (\text{ind}_N p p0 ps n)
\end{array}
\]

where \( p : \text{N}\Rightarrow\text{Prop} \) is a predicate on \( N \) (\( \text{Prop} \) 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 [11]. As a follow-up, one could consider how to handle co-inductive types [5] and co-recursion [1] in Dedukti as well, that is, terms representing infinite objects like streams.
**Workplan.** One can start by considering the case of a simple first-order inductive type like \( \mathbb{N} \), and then consider increasingly more complex classes of inductive types:

- **polymorphic inductive types**

  ```
  inductive L : Set \Rightarrow TYPE := // lists
  | nil(\alpha) : L\alpha
  | cons(\alpha) : \tau\alpha \Rightarrow L\alpha \Rightarrow L\alpha
  ```

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

- **dependent inductive types**

  ```
  inductive V : Set \Rightarrow \mathbb{N} \Rightarrow TYPE := // vectors
  | nil(\alpha) : V\alpha 0
  | cons(\alpha) : \tau\alpha \Rightarrow \forall n, V\alpha n \Rightarrow V\alpha (s n)
  ```

- **mutually defined inductive types**

  ```
  inductive tree : TYPE := // trees with finite branching
  | leaf : tree
  | node : forest \Rightarrow tree
  and forest : TYPE :=
  | empty : forest
  | cons: tree \Rightarrow forest \Rightarrow forest
  ```

- **strictly positive inductive types**

  ```
  inductive O : TYPE := // ordinals
  | 0
  | s : 0 \Rightarrow 0
  | sup : (\mathbb{N} \Rightarrow 0) \Rightarrow 0
  ```

- **positive inductive types**

  ```
  inductive Rou : TYPE := // continuations
  | over : Rou
  | next : ((Rou \Rightarrow \mathbb{L}N) \Rightarrow \mathbb{L}N) \Rightarrow Rou
  ```

- **nested inductive types**

  ```
  inductive Bush : Set \Rightarrow TYPE :=
  | nil(\alpha) : Bush\alpha
  | cons(\alpha) : \tau\alpha \Rightarrow Bush (Bush\alpha) \Rightarrow Bush\alpha
  ```

- **inductive predicates**

  ```
  inductive \leq : \mathbb{N} \Rightarrow \mathbb{N} \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)
  ```

2
• inductive-inductive types [9]

```ocaml
ductive SortL : TYPE := // sorted lists
| nil : SortL
| cons : ∀ y l (p : Le y l), SortL
and Le : N ⇒ SortL ⇒ TYPE :=
| Le_nil : ∀ x, Le x nil
| Le_cons : ∀ x y l (p : Le y l),
            x ≤ y ⇒ Le x l ⇒ Le x (cons y l p)
```

• inductive-recursive definitions [6]

```ocaml
ductive UniqL : TYPE := // lists with unique elements
| nil : UniqL
| cons : ∀ x l, x /∈ l ⇒ UniqL
and /∈ : N ⇒ UniqL ⇒ TYPE
modulo x /∈ nil → True
       x /∈ cons y l _ → x ≠ y ∧ x /∈ l
```

• the combination of both [7]

• quotient types [8]

```ocaml
ductive Bag : TYPE := // multisets
| nil : Bag
| cons : N ⇒ Bag ⇒ Bag
modulo
| swap : ∀ x y b, cons x y b = cons y x b
```

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

• higher dimensional inductive types [12]

```ocaml
ductive S2 : TYPE := // homotopic sphere
| base : S2
| surf : eq (eq base base) refl refl
modulo
```

where eq is Leibniz equality and refl the canonical reflexivity proof.

• co-inductive types [1]

```ocaml
cinductive stream : TYPE // streams of natural numbers
| head : stream ⇒ N
| tail : stream ⇒ stream
```

```ocaml
constant 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**


