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 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:

```
inductive Nat: TYPE ≔ // natural numbers
| 0: Nat
| s: Nat ⇒ Nat
```

we would like the following code to be internally generated:

```
constant symbol Nat: TYPE
constant symbol 0: Nat
constant symbol s: Nat ⇒ Nat
symbol ind_Nat: ∀p,π(p0) ⇒ (∀n,π(pn) ⇒ π(p(sn))) ⇒ ∀n,π(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)
```

where `p:Nat⇒Prop` is a predicate on `Nat` (`Prop` is the type of propositions) and `π:Prop⇒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 [5] 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 `Nat`, and then consider increasingly more complex classes of inductive types:

- polymorphic inductive types

  ```
  inductive List: Set ⇒ TYPE ≔ // lists
  | nil{α}: List α
  | cons{α}: τ α ⇒ List α ⇒ List α
  ```

  where `Set` is the type of sets and τ:`Set⇒TYPE` maps sets to types.

- dependent inductive types

  ```
  inductive V: Set ⇒ Nat ⇒ TYPE ≔ // vectors
  | nil{α}: Vec α 0
  | cons{α}: τα ⇒ ∀n,Vec α n ⇒ Vec α (s n)
  ```

- mutually defined inductive types

  ```
  inductive Tree: TYPE ≔ // trees with finite branching
  | leaf: Tree
  | node: Forest ⇒ Tree
  and Forest: TYPE ≔
  | empty: Forest
  | cons: Tree ⇒ Forest ⇒ Forest
  ```

- strictly positive inductive types

  ```
  inductive Ord: TYPE ≔ // ordinals
  | 0: Oord
  | s: Ord ⇒ Ord
  | sup: (Nat ⇒ Ord) ⇒ Ord
  ```

- positive inductive types [4]

  ```
  inductive Rou: TYPE ≔ // continuations
  | over: Rou
  | next: ((Rou ⇒ List nat) ⇒ List nat) ⇒ Rou
  ```

- nested inductive types [13]

  ```
  inductive Bush: Set ⇒ TYPE ≔
  | nil{α}: Bush α
  | cons{α}: τ α ⇒ Bush (Bush α) ⇒ Bush α
  ```

- inductive predicates

```
inductive ≤: Nat ⇒ Nat ⇒ Prop ≔
| 0≤ : ∀x,π(0 ≤ x)
| s≤ : ∀x y,π(x ≤ y) ⇒ π(s x ≤ s y)
```

- inductive-inductive types [11]

```
inductive SortL: TYPE ≔ // sorted lists
| nil:SortL
| cons: ∀y l (p:Le y l), SortL
and Le: Nat ⇒ 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 [8]

```
inductive UniqL: TYPE ≔ // lists with unique elements
| nil: UniqL
| cons: ∀x l, x ∉ l ⇒ UniqL
and ∉: Nat ⇒ UniqL ⇒ TYPE
modulo x ∉ nil          → True
       x ∉ cons y l _ → x ≠ y ∧ x ∉ l
```

- the combination of both [9]
- quotient types [10]

```
inductive Bag: TYPE ≔ // multisets
| nil: Bag
| cons: Nat ⇒ 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 [16]

```
inductive S2: TYPE ≔ // homotopic sphere
| base: S2
modulo
| surf: eq (eq base base) refl refl
```

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

- cyclic data types [12]

```
inductive CList : TYPE // cyclic lists
| nil : CList
| cons : Nat → CList → CList
modulo ... // the axioms of cyclicity
```

- co-inductive types [1]

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

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

[1] A. Abel and B. Pientka. Well-founded recursion with copatterns and sized types. *Journal of Functional Programming*, 26(e2), 2016.

[2] T. Altenkirch and A. Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages*, 2016.

[3] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Dedukti: a logical framework based on the $\lambda\Pi$-calculus modulo theory, 2019. Draft.

[4] U. Berger, R. Matthes, and A. Setzer. Martin Hofmann's Case for Non-Strictly Positive Data Types. In *Proceedings of the 24th International Conference on Types for Proofs and Programs*, Leibniz International Proceedings in Informatics 130, 2019.

[5] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly Modular (Co)datatypes for Isabelle/HOL. In *Proceedings of the 5th International Conference on Interactive Theorem Proving*, Lecture Notes in Computer Science 8558, 2014.

[6] F. Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005.

[7] H. Comon. Inductionless induction. In *Handbook of Automated Reasoning. Volume I*, chapter 14, pages 913–962. Elsevier, 2001.

[8] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549, 2000.

[9] P. Dybjer and A. Setzer. Indexed induction-recursion. *Journal of Logic and Algebraic Programming*, 66(1):1–49, 2006.

[10] M. P. Fiore, A. M. Pitts, and S. C. Steenkamp. Constructing infinitary quotient-inductive types. `https://arxiv.org/abs/1911.06899`, 2019.

[11] F. Nordvall Forsberg and A. Setzer. A finite axiomatisation of inductive-inductive definitions. In U. Berger, H. Diener, P. Schuster, and M. Seisenberger, editors, *Logic, Construction, Computation*, volume 3 of *Ontos Mathematical Logic*, pages 259–288. De Gruyter, 2012.

[12] M. Hamana. Cyclic Datatypes modulo Bisimulation based on Second-Order Algebraic Theories. *Logical Methods in Computer Science*, 13(4):1–38, 2017.

[13] P. Johann and A. Polonsky. Higher-Kinded Data Types: Syntax and Semantics. In *Proceedings of the 34th IEEE Symposium on Logic in Computer Science*, 2019.

[14] K. Kikuchi and T. Aoto I. Sasano. Inductive Theorem Proving in Non-terminating Rewriting Systems and Its Application to Program Transformation. In *PPDP19*.

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

[16] The Univalent Foundation Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Studies, 2013.

[17] D. Walukiewicz-Chrząszcz and J. Chrząszcz. Consistency and completeness of rewriting in the calculus of constructions. *Logical Methods in Computer Science*, 4(3:8):1–20, 2008.