Performing Implicit Induction Reasoning with Certifying Proof Environments

Amira Henaien, Sorin Stratulat

Performing Implicit Induction Reasoning with Certifying Proof Environments

Amira Henaien[1,2] Sorin Stratulat[1]

[1] LITA, Université de Lorraine,
Ile du Saulcy, 57000, Metz, France
{amira.henaien,sorin.stratulat}@univ-lorraine.fr
[2] Higher School of Communication of Tunis (Sup'Com), University of Carthage, Tunisia

Abstract. Largely adopted by proof assistants, the conventional induction methods based on explicit induction schemas are non-reductive and local, at schema level. On the other hand, the implicit induction methods used by automated theorem provers allow for lazy and mutual induction reasoning. In this paper, we present a new tactic for the Coq proof assistant able to perform automatically implicit induction reasoning. By using an automatic black-box approach, conjectures intended to be manually proved by the certifying proof environment that integrates Coq are proved instead by the Spike implicit induction theorem prover. The resulting proofs are translated afterwards into certified Coq scripts.

1 Introduction

Unbounded data structures like naturals and lists are abundant in today's first-order specifications. Issued from the general principles of Noetherian induction and its counter-positive version, the *Descente Infinie* induction, various proof techniques have been devised to effectively reason on well-founded posets. Given such a poset, the soundness of the reasoning is guaranteed by the underlying well-founded ordering which forbids infinite strictly decreasing sequences of elements from the poset.

In a first-order setting, one can distinguish two classes of Noetherian induction methods. The first class, of conventional induction methods, is based on explicit induction schemas [7]. An induction schema can attach to a formula, called *induction conclusion*, a set of formulas defined as *induction hypotheses* (IHs) to be used exclusively by the offspring of the induction conclusion in the proof derivation. The induction orderings are defined locally, at schema level, and can differ from one schema to another. Such methods are widely-spread among proof assistants because they can be easily integrated into sequent-based inference systems as a separate inference rule. On the other hand, it is not lazy, so it may happen that the IHs be defined (sometimes long) before their use or to not be exploited at all. The main challenges to overcome are the definition of the right IHs followed by the guidance of the proof development related to the induction conclusions in order to properly use the defined IHs. Moreover, since

only the offspring of the induction conclusion can use its attached IHs, the mutual induction with other formulas from the proof cannot be done naturally. The second class, of implicit induction techniques, allows for lazy and mutual induction because any formula instance from the proof can be used as IH, as long as it is smaller than (and sometimes equal to) the current conjecture. They fit better for reasoning as a working mathematician [25]. On the other hand, the proof derivations are reductive, requiring that the ground instances of newly derived conjectures in the proof be smaller than (and sometimes equal to) some ground instances of the current conjecture. In order to satisfy the ordering constraints, the induction ordering is global, at proof level. These techniques are originating from the Knuth-Bendix completion algorithm [13,15] and are usually integrated into automatic theorem provers.

Proofs of similar conjectures done with the Spike implicit induction prover [3,19,2] and with proof assistants like Coq [23] and PVS [18] have been previously compared [17,2]. We have witnessed that many of the conjectures can be automatically proved by Spike, and concluded that the number of user interactions can dramatically reduce if the proof assistants integrate implicit induction techniques. To achieve this goal, we present a new Coq tactic that performs lazy and mutual induction reasoning directly from the certifying proof environment provided by Coq. In an automatic way, subgoals from a Coq script are exported to be proved by Spike. The resulting Spike proofs are translated back into certified Coq scripts, as shown by previous works [20,22].

Related works In the past, a lot of effort has been put into adding implicit induction features to explicit induction proofs. In [5,12,4], explicit induction-based proof techniques systems have been extended to deal with certain classes of mutually defined recursive functions. Also, Protzen [16] has defined a proof strategy to perform lazy induction on particular explicit induction proofs. In the other direction, Courant [9] identified a class of implicit induction proofs that can be reconstructed into conventional induction proofs. More recently, Voicu and Li [24] proposed a *Descente Infinie* tactic in Coq that identifies repeated subgoals in a Coq script by analyzing their (partial) proof terms. A recursive function definition and an associated explicit induction schema are issued from the analysis process. The implementation of the tactic is rather limited, requiring (a restricted form of) rewriting due to the reductive nature of the *Descente Infinie* induction-based proof methods. A naïve automation algorithm was presented for simple inductive proofs.

Structure of the paper The paper has 5 sections and one appendix. After the introductory section, Section 2 introduces the main induction principles adapted for first-order reasoning and shows how the implicit induction technique is implemented in Spike. Section 3 briefly presents the Coq system and describes the automatic tactic. The implementation details of the tactic and some experimental results are given in Section 4. The last section concludes. The statistics about the experimental part and the workflow of the `Spike` tactic are given in the appendix.

2 Induction principles and the Spike theorem prover

Induction principles The induction principles of interest are instances of the general Noetherian principle adapted for first-order logic. In [21], they have been qualitatively distinguished according to the kind of elements we are reasoning on, which can be either terms or first-order formulas.

The *term-based induction* principle considers that, in order to prove a property ϕ over a set of (vectors of) terms $\mathcal{E}$, it is enough to prove it for each element, knowing that we have the right to assume as IH the fact that ϕ is true for any smaller element. The explicit induction proof techniques, like the structural induction [7], implement the term-based induction principle, hence they can be applied to prove only one property, in our case ϕ.

Definition 1 (term-based induction). $(\forall$ *term vector* $\overline{m} \in \mathcal{E}, (\forall$ *term vector* $\overline{k} \in \mathcal{E}, \overline{k} < \overline{m} \Rightarrow \phi(\overline{k})) \Rightarrow \phi(\overline{m})) \Rightarrow \forall$ *term vector* $\overline{p} \in \mathcal{E}, \phi(\overline{p})$.

On the other hand, several properties can be proved simultaneously by using the *formula-based induction* principle if the elements are instead first-order formulas from a set $\mathcal{E}'$. The implicit induction is an example of formula-based induction. It allows to a formula to be used as IH in the proof of another formula as long as it is smaller, hence it is able to perform mutual induction lazily.

Definition 2 (formula-based induction). $(\forall$ *formula* $\delta \in \mathcal{E}', (\forall$ *formula* $\gamma \in \mathcal{E}', \gamma < \delta \Rightarrow \gamma) \Rightarrow \delta) \Rightarrow \forall$ *formula* $\rho \in \mathcal{E}',$ ρ.

The soundness of the two inductive principles is ensured if the induction ordering is well-founded, i.e., in our case, no infinite strictly decreasing elements of $\mathcal{E}$ and $\mathcal{E}'$, respectively, can be built.

The Spike theorem prover Spike [3,19,2] is a first-order theorem prover that can prove properties about conditional specifications consisting of a set of axioms represented by (conditional) equalities. In the past, Spike has been used to automatize the validation process of several non-trivial applications as the JavaCard Platform [2] and a telecommunications protocol [17].

Its inference system is able to build implicit induction proofs of the properties given as initial conjectures by using inference rules that replace a conjecture from the current state of the proof with a (potentially empty) set of new conjectures. A Spike proof of a non-empty set of conjectures E^0 is a finite sequence of states of the form $(E^0, \emptyset) \vdash (E^1, H^1) \vdash \cdots \vdash (H^{n-1}, H^{n-1}) \vdash (\emptyset, H^n)$, where $E^i (i \in [0..n-1])$ are multisets of conjectures and $H^i (i \in [1..n])$ are multisets of previously treated conjectures.

The formula-based induction principle implemented by Spike considers $\mathcal{E}'$ as the set of all ground instances of the conjectures encountered in a proof but the IHs to be used during a proof step are instances of formulas only from the current state. The induction ordering is globally defined, at proof level. Moreover, the inference rules are *reductive* such that for each ground instance of the new conjectures generated in a proof step, there should exist a smaller (and sometimes ordering-equivalent) logically equivalent instance of a formula from the current state.

3 Calling the Spike Prover from a Coq Script

The Coq proof assistant Coq [23] is a proof assistant based on type theory. Its specifications can be written in different ways. One of them is the functional style, using pattern matching and recursion. In this case, the admissible functions have to be well-typed and well-founded. The well-foundedness property of a function can be checked by syntactical constraints on one of the arguments of the function, called *decreasing argument*. When this argument is not explicitly stated, Coq tries to determinate it but, in most of the cases, it is better to be given by the user. This can be problematic when the functions are mutually recursively defined and the user has to provide the induction ordering.

Example In the following, we propose a Coq specification of the mutually recursive functions *even, oeven, odd* and *eodd* that take as arguments a natural and return a boolean, using the Coq construction `Fixpoint ... with`. The functions *even* and *oeven* (resp. *odd* and *eodd*) are different implementations of the predicate checking whether a natural is even (resp. odd). The constructor symbols for the naturals are *0* and the successor S and for booleans are *true* and *false*. The constructor symbols are free, i.e., there is no equality relation between them, and help to define structural induction schemas from the function definitions. They allow to clearly separate different branches in the function definitions, using the `match...with` pattern matching construction.

```
Fixpoint even (x: nat) :bool :=
  match x with
  | 0 ⇒ true
  | S 0 ⇒ false
  | S (S n) ⇒ oeven n
end
  with oeven (x: nat) : bool :=
    match x with
  | 0 ⇒ true
  | S 0 ⇒ false
  | S (S n) ⇒ if odd n then false
                else even n
end
  with odd (x: nat) : bool :=
    match x with
  | 0 ⇒ false
  | S 0 ⇒ true
  | S (S n) ⇒ eodd n
end
  with eodd (x: nat) : bool :=
    match x with
  | 0 ⇒ false
  | S 0 ⇒ true
  | S (S n) ⇒ if even n then odd n
                else true
end.
```

Let us prove the theorem

```
Theorem even_xx: ∀ x, even (add x  x) = true.,
```

where *add* is the usual addition operator:

```
Fixpoint add (x y: nat) :nat:=
match x with                    | S u ⇒ S(add u y)
| 0 ⇒ y                         end.
```

It can be shown that the trivial explicit induction schema which replaces successively x by 0 and $S(y)$ does not work to successfully finish the proof of the theorem. New user interaction is required, for example, by adding other induction steps. The generation of useful induction schemas is not trivial for the general case. If the user has no idea how the proof will be performed, the success of the proof attempt may depend on:

- the set of induction variables from the current conjecture,
- the way the induction variables are instantiated. In our example, we have chosen 0 and S(n), but we could have had also 0, S(0) and S(S(n)), or 0, S(0), S(S(0)) and S(S(S(n))), etc. (in fact, there is an unbounded number of possibilities), and
- the set of IHs attached to each induction conclusion and how they are applied further in the proof. It may happen that additional IHs to be defined but not used, or to miss crucial IHs in the definition of induction schemas.

A more detailed discussion about the 'proof by induction' problems and challenges can be found in [11].

The `Spike` *tactic* An automatic solution that we propose to the Coq users is to call instead the Spike theorem prover. After providing two lemmas about some simple `add` properties, the proof can be completely done with the `Spike` tactic:

```
Spike [ordering constraints]
```

The tactic firstly generates a Spike specification from the analysis of the Coq script, then Spike is executed to prove the theorem using an induction ordering based on precedencies over the function symbols given as arguments to the tactic. The precedencies can define equivalence and strict ordering relations. In our case, the tactic and its arguments are:

```
Spike equiv [[even, oeven, odd, eodd]]
      greater [[even, true, false, S, 0, add], [add, S, 0]].
```

It indicates to Spike that the symbols *even, oeven, odd, eodd* are equivalent and that *even* (resp. *add*) is greater than *true, false, S, 0*, and *add* (resp. *S* and *0*). The implementation details of the `Spike` tactic are given in the next section.

4 The Implementation of the `Spike` Tactic

Tactics are at the heart of building proofs in Coq. They may use elements of the current context of a given proof, as declarations, definitions, axioms, hypotheses,

lemmas and already proved theorems to solve the current goal. Coq allows the integration of new tactics by two means: either by Ltac [10], a tactical language for Coq, or by the OCAML programming language [14]. The first solution is less automated because of the difficulty to directly call an external program, in occurrence the Spike prover, and of file reading and writing operations. The second solution is based on an interface between Coq and OCAML which lets developers integrate their own tactics written in the fully featured OCAML language. On the other hand, no safety control mechanisms are provided. Hopefully, the proof terms built from tactics can be type-checked by the Coq kernel at the end of a proof script. In addition, the lack of documentation and the 'not sufficiently explained' Coq source code usually make the implementation task difficult. The second solution will be detailed for proving *even_xx*.

Firstly, we present the full Coq script representing the axiomatic translation of the function-based Coq script from Section 3 and the proof of *even_xx*:

```
Axiom add1 : ∀ x, add 0 x = x.
Axiom add2 : ∀ x y, add (S x) y = S( add x y).

Axiom even1 : even 0 = true.
Axiom even2 : even (S 0) = false.
Axiom even3 : ∀ x, even (S (S x)) = oeven x.

Axiom oeven1 : oeven 0 = true.
Axiom oeven2 : oeven (S 0) = false.
Axiom oeven3 : ∀ x, odd x = true → oeven (S (S x)) = false.
Axiom oeven4 : ∀ x, odd x = false → oeven (S (S x)) = even x.

Axiom odd1 : odd 0 = false.
Axiom odd2 : odd (S 0) = true.
Axiom odd3 : ∀ x, odd (S (S x)) = eodd x.

Axiom eodd1 : eodd 0 = false.
Axiom eodd2 : eodd (S 0) = true.
Axiom eodd3 : ∀ x, even x = true → eodd (S (S x)) = odd x.
Axiom eodd4 : ∀ x, even x = false → eodd (S (S x)) = true.


Theorem addx0: ∀ x , add x 0 = x.
Spike equiv [[even, oeven, odd, eodd]] greater [[oeven, add, S, 0, true, false],
[add, S, 0]]. Qed.

Theorem addS1: ∀ x y, add x (S y) = S (add x y).
Spike equiv [[even, oeven, odd, eodd]] greater [[oeven, add, S, 0, true, false],
[add, S, 0]]. Qed.

Theorem even_xx: ∀ x, even(add x x) = true.
Spike equiv [[even, oeven, odd, eodd]] greater [[oeven, add, S, 0, true, false],
[add, S, 0]]. Qed.
```

The soundness of the translation can be checked for each axiom, represented as a new theorem which is further proved. For example, the axiom *even3* can be validated by the theorem:

```
Theorem even3_soundness : ∀ x, even (S (S x)) = oeven x.
intro x. simpl. trivial. Qed.
```

The `Spike` tactic comes into four variants:

Spike - the ordering constraints are inferred by Spike directly from the specification.

Spike equiv [$S_1, \ldots, S_n$**]** - each S_i ($i \in [1..n]$) is a set of equivalent symbols.

Spike greater [$S_1, \ldots, S_n$**]** - each S_i ($i \in [1..n]$) is of the form $\{symb_1, symb_2, \ldots, symb_n\}$ such that $symb_1$ is greater than any of the symbols from the set $\{symb_2, \ldots, symb_n\}$.

Spike equiv [$S_1, \ldots, S_n$ **] greater [** $S_1^{'}, \ldots, S_n^{'}$ **]** is the combination of the two previous cases.

The tactic starts by extracting the Spike specification and the conjecture to be proved from the current section, then calls Spike to generate an implicit induction proof of the conjecture. If successful, a Coq script is generated from the Spike proof representing the Coq proof of an equivalent theorem to the current one. The script is certified by the Coq kernel, then the equivalent theorem is applied to solve the original Coq conjecture. The proof is completed only after the Coq kernel has successfully type-checked the whole proof, i.e., when the `Qed` command ending the proof is executed. We will detail each step of the tactic for the proof of *even_xx*, schematised in Fig. 1 from the appendix.

Extracting the Spike specification A Spike specification consists of two parts. The first one describes the sorts with their constructors and the defined function symbols by means of conditional equalities playing the role of axioms. The second part influences the way the proof will be performed. It includes the definition of the precedencies over the function symbols, the definition of the inference rules and the proof strategy. It also includes the definition of the conjectures to be proved. The `Spike` tactic is able to extract all this information from the current Coq proof environment by the means of the "*spike*" OCAML module. Any Coq specification that intends to use the `Spike` tactic has to declare previously this module, as follows:

```
Declare ML Module "spike".
```

The Coq specification should define the functions axiomatically. One solution is to extract the axioms directly from the fixpoint definitions, then validate them as shown in the previous section for the axiom *even3*. The tactic starts by identifying the conjecture to be proved from the current goal, then translates all axioms existing in the current context, together with the sorts, function symbols

and constructors. All definitions and conjectures are adapted to be accepted syntactically in Spike. In order to ease the translation from Spike proofs to Coq scripts process, the generated Spike specifications may contain inline Coq scripts, for example declaring the signature of the function symbols using `Parameter`. For our example, the Spike specification starts with:

sorts: bool nat ;
constructors:
true : → bool;
false : → bool;
0 : → nat;
S_: nat → nat;
defined functions:

even_: nat → bool;	$ `Parameter` even : nat → bool.
oeven_: nat → bool;	$ `Parameter` oeven : nat → bool.
odd_: nat → bool;	$ `Parameter` odd : nat → bool.
eodd_: nat → bool;	$ `Parameter` eodd : nat → bool.
add__: nat nat → nat ;	$ `Parameter` add : nat → nat → nat.

Translating the Spike proof into Coq script The Spike prover is automatically executed on the generated specification using a mode that can produce Coq script from a proof, as shown in [20,22]. In the following, we only recall the key steps of the translation process. In order to reproduce the implicit induction reasoning, for each formula F a comparison weight W is associated. When F is instantiated, its weight has to be instantiated in the same way. This is achieved by factorizing variables using functionals of the form (`fun` $\overline{x} \Rightarrow$ *(F,W)*), where $\overline{x}$ is a vector of variables shared between F and W. For instance, the functional associated to the conjecture labelled by 91 in the Spike proof and being the equivalent of our theorem *even_xx* is:

`Definition` *type_LF_91* := nat → (Prop * (List.list term)).
`Definition F_91` : *type_LF_91*:= (fun u1 ⇒ ((even (add u1 u1)) = true, (Term id_even ((Term id_add ((model_nat u1):: (model_nat u1)::nil))::nil))::(Term id_true nil)::nil)).

The induction orderings used by Spike are syntactic and exploit the tree representation of terms. In Coq, we need to abstract the Coq formulas into multisets of special terms built from a term algebra provided by the COCCINELLE library [8]. For example, 'id_even' is the COCCINELLE equivalent of 'even'. There are also the *model* functions that translate Coq terms into COCCINELLE terms. Finally, we get two main parts: firstly, the specification part contains the current context with some informations provided to COCCINELLE in order to create operational term algebra. It includes other important steps, like the description of the induction ordering and its properties, as well as the soundness proof of the underlying Noetherian induction principle. Secondly, the direct one-to-one translation of the Spike proof for which the application of every inference rule has a corresponding Coq script.

An important contribution with respect to the previous works is the replacement of the fixpoint-based functions by axioms. This improvement allowed a more effective one-to-one translation of many Spike rule applications, in particular for those dealing with conditional equalities. In the previous works, the method to identify the conditional axiom used by a Spike rule consists in unfolding the fixpoint definition of the function symbol defined by the conditional axiom, then choosing the branch of the fixpoint definition that validates the conditions of the axiom in the current proof context. The generation of Coq script for the validation part is not fully automatisable, so it may lead to the failure of the `Spike` tactic. Moreover, thanks to the axiomatic representation, many of the unconditional axioms can now participate to automatize even more the translation process. Different rewriting operations have been simplified:

i) unconditional rewriting of the current goal: any unconditional axiom can be added to a rewriting base. For our example, the corresponding script is:

   `Hint` *Rewrite addS1 addx0 add1 add2 even1 even2 even3 oeven1 oeven2 odd1 odd2 odd3 eodd1 eodd2*: *rewrite_axioms*.
   `Ltac` *rewrite_ax*:= *autorewrite* `with` *rewrite_axioms*.

   During the translation process, any application of unconditional rewriting rule in the implicit proof will be replaced by *rewrite_ax*.
ii) conditional rewriting of the current goal: it is translated into *rewrite name_of_conditional_axiom*.
iii) unconditional rewriting of a condition H of the current goal: it is translated into *normalize* `with` *rewrite_axioms* `in` *H*, where *normalize* is a new Coq tactic we defined in order to rewrite H with the rewriting base *rewrite_axioms*.

The theorem equivalent to *even_xx* is:

`Theorem` *true_91:* $\forall$ *u1, (even (add u1 u1)) = true.*,

excepting that it is expected to be proved in the context of the function symbols given as parameter to the Coq section including *true_91*. Once the Coq script has been successfully checked, it is imported into the current environment as a library. Besides that, if a theorem needs some other theorems to be proved in terms of lemmas, they should be put in the same section. The `Spike` strategy will import the Coq script, apply the theorem *true_91*, then discharge the parameters with function symbols from the original Coq specification. The corresponding Coq script is:

```
Require Import "Coq script with the proof of true_91".
apply true_91;
repeat (apply even | apply oeven | apply odd | apply eodd | apply add).
```

Extensions. The `Spike` tactic can only define ordering constraints for the current proof. In order to deal with more complex proofs, two extensions are proposed:

– `SpikeWithFullind_aug [ordering constraints]` to integrate the augment technique into the proof strategy. Presented in [6] and previously implemented in Spike [1], it allows to increase the factual base from the context of the simplifying conjectures.
– `SpikeWithIndPriorities [induction strategy][ordering constraints]` to decide which terms from a conjecture can be instantiated. The induction strategy given as argument establishes a priority among the function symbols that may occur at root positions of these terms.

Statistics. We have also used the `Spike` tactic and its extensions for other examples treated in previous works [20,22]. Table 1 illustrates the number of lemmas (i.e., previously proved conjectures), hypotheses (i.e., not yet proved conjectures), the employed tactic and whether it has used parameters or not, as well as the execution time for some successfully proved conjectures involved in the validation of a telecommunications protocol [17]. `Aug` and `Ind` denote the use of `SpikeWithFullind_aug` and `SpikeWithIndPriorities` tactics, respectively. The proofs have been performed with a MacBook Pro featuring 4GB of RAM and a 2-core Intel processor i5 at 2,5 GHz. The full Coq scripts can be found on Spike's website http://code.google.com/p/spike-prover/. [3]

5 Conclusions and Future Works

We have presented the `Spike` tactic and some of its extensions to automatically perform implicit induction by the Coq proof assistant. Using a black-box approach, Coq is now able to certify non-trivial conjectures whose proofs may require multiple induction steps, as well as lazy and mutual induction reasoning. As a case study, conjectures involved in the validation of a non-trivial application have been successfully and directly certified by Coq using the `Spike` tactic. The proofs of more than 60% of them have been performed completely automatically, i.e., the Coq user does not need to provide any argument to the tactic. On the other hand, its application is limited to Coq specifications transformable into conditional specifications whose axioms can be oriented into rewrite rules.

In the near future, we intend to automatize the translation process of a fixpoint-based function definition into axioms, as well as to define a set of Coq tactics that can build implicit inductive proofs directly into Coq either automatically, for example by simulating Spike's behaviour, or interactively. For the last case, a proof environment has to be designed to facilitate the user's access to crucial information from the whole proof, in particular from the proof branches other than that corresponding to the current goal. The idea is to use this information to discover lazily the IHs allowing to perform mutual induction reasoning. In a longer perspective, we will test whether these ideas can be extrapolated to cyclic proofs and applied to the recent induction methods proposed in [21].

[3] Other examples of Coq scripts using the `Spike` tactic can be accessed from the website.

References

1. A. Armando, M. Rusinowitch, and S. Stratulat. Incorporating decision procedures in implicit induction. *J. Symb. Comput.*, 34(4):241–258, 2002.
2. G. Barthe and S. Stratulat. Validation of the JavaCard platform with implicit induction techniques. In R. Nieuwenhuis, editor, *RTA*, volume 2706 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2003.
3. A. Bouhoula, E. Kounalis, and M. Rusinowitch. Automated mathematical induction. *Journal of Logic and Computation*, 5(5):631–668, 1995.
4. R. Boulton and K. Slind. Automatic derivation and application of induction schemes for mutually recursive functions. In J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. Pereira, Y. Sagiv, and P. Stuckey, editors, *Computational Logic — CL 2000*, volume 1861 of *Lecture Notes in Computer Science*, pages 629–643. Springer Berlin / Heidelberg, 2000.
5. R. S. Boyer and J S. Moore. *A computational logic handbook*. Academic Press Professional, 1988.
6. R. S. Boyer and J S. Moore. *Integrating decision procedures into heuristic theorem provers: a case study of linear arithmetic*. Oxford University Press, Inc. New York, NY, USA, 1988.
7. R. M. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12:41–48, 1969.
8. E. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Certification of automated termination proofs. *Frontiers of Combining Systems*, pages 148–162, 2007.
9. J. Courant. Proof reconstruction. Research Report RR96-26, LIP, 1996. Preliminary version.
10. D. Delahaye. A tactic language for the system Coq. In M. Parigot and A. Voronkov, editors, *Logic for Programming and Automated Reasoning (LPAR)*, volume 1955 of *Lecture Notes in Computer Science (LNCS)*, pages 85–95, Reunion Island (France), November 2000. Springer.
11. B. Gramlich. Strategic issues, problems and challenges in inductive theorem proving. *Electronic Notes in Theoretical Computer Science*, 125(2):5–43, March 2005.
12. D. Kapur and M. Subramaniam. Automating induction over mutually recursive functions. In *Algebraic Methodology and Software Technology*, volume 1101 of *LNCS*, pages 117–131. Springer, 1996.
13. D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–297, 1970.
14. X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system - release 3.12. Documentation and user's manual*. INRIA.
15. D. R. Musser. On proving inductive properties of abstract data types. In *POPL*, pages 154–162, 1980.
16. M. Protzen. Lazy generation of induction hypotheses. *Automated Deduction — CADE-12*, pages 42–56, 1994.
17. M. Rusinowitch, S. Stratulat, and F. Klay. Mechanical verification of an ideal incremental ABR conformance algorithm. *J. Autom. Reasoning*, 30(2):53–177, 2003.
18. N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS prover guide - version 2.4*. SRI International, November 2001.
19. S. Stratulat. A general framework to build contextual cover set induction provers. *J. Symb. Comput.*, 32(4):403–445, 2001.

20. S. Stratulat. Integrating implicit induction proofs into certified proof environments. In *Integrated Formal Methods*, volume 6396 of *Lecture Notes in Computer Science*, pages 320–335, 2010.
21. S. Stratulat. A unified view of induction reasoning for first-order logic. In A. Voronkov, editor, *Turing-100*, volume 10 of *EPiC Series*, pages 326–352. EasyChair, 2012.
22. S. Stratulat and V. Demange. Automated certification of implicit induction proofs. In *CPP'2011 (First International Conference on Certified Programs and Proofs)*, volume 7086 of *Lecture Notes Computer Science*, pages 37–53. Springer-Verlag, 2011.
23. The Coq Development Team. The Coq reference manual - version 8.2. http://coq.inria.fr/doc, 2009.
24. R. Voicu and M. Li. Descente Infinie proofs in Coq. In *The 1st Coq Workshop*, page 12 pages, 2009.
25. C.-P. Wirth. Descente infinie + Deduction. *Logic Journal of the IGPL*, 12(1):1–96, 2004.

A Tables and Figures

# Name	Lemmas	Hypotheses	Tactic	Parameters	Time (s)
1. firstat_timeat	1	0	Spike	no	3.021
2. firstat_progat	1	0	Spike	no	3.159
3. sorted_sorted	0	0	Spike	no	2.295
4. sorted_insat1	4	0	Aug	yes	14.233
5. sorted_insin2	4	0	Aug	yes	15.400
6. sorted_e_two	0	0	Spike	no	2.039
7. member_t_insin	2	0	Spike	yes	8.558
8. membert_insat	3	0	Spike	no	11.317
9. member_firstat	2	0	Spike	no	8.471
10. timel_insat	0	0	Spike	no	2.402
11. erl_insin	0	0	Spike	yes	2.512
12. erl_insat	0	0	Spike	no	2.488
13. erl_prog	2	0	Spike	yes	11.768
14. time_progat_er	1	0	Spike	no	4.238
15. timeat_tcrt	0	0	Spike	no	3.225
16. timel_timeat_max	2	1	Aug	yes	9.191
17. null_listat	1	0	Spike	no	4.143
18. null_listat1	0	0	Spike	no	1.958
19. cons_insat	0	0	Spike	no	1.987
20. cons_listat	0	0	Spike	no	2.004
21. progat_timel_erl	3	0	Aug	yes	11.621
22. progat_insat	3	1	Aug	yes	44.422
23. progat_insat1	3	0	Aug	yes	17.055
24. timel_listupto	0	0	Spike	no	2.295
25. sorted_listupto	3	0	Aug	yes	13.042
26. time_listat	1	0	Spike	no	5.720
27. sorted_cons_listat	3	1	Ind	yes	16.699
28. null_wind2	0	1	Spike	no	3.382
29. timel_insin1	1	0	Spike	yes	4.456
30. null_listupto1	0	0	Spike	no	1.949
31. erl_cons	0	0	Spike	no	2.415
32. no_time	1	1	Spike	no	7.129
33. final	2	1	Spike	no	8.330

Table 1. Statistics about the ABR proofs.

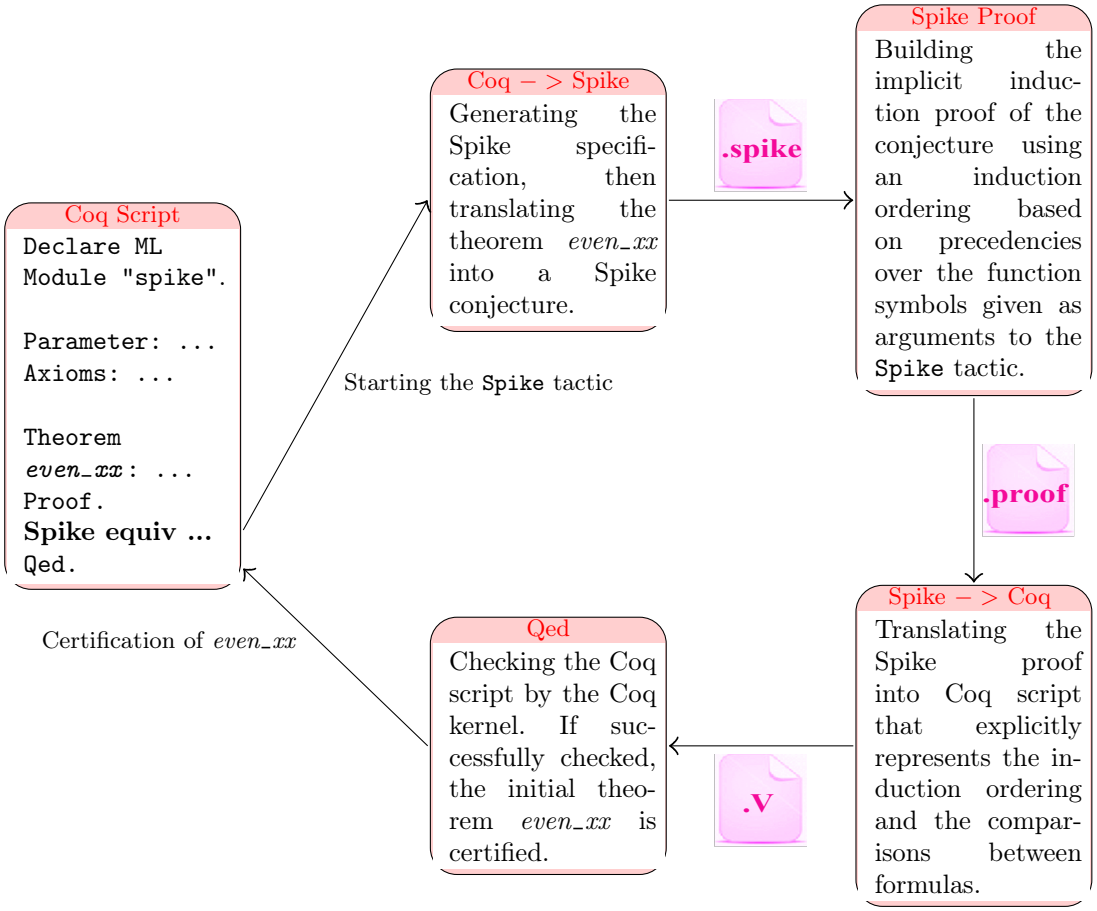

Fig. 1. The integration workflow of the *even_xx* Spike proof into Coq using the `Spike` tactic.