Integrating Implicit Induction Proofs into Certified Proof Environments

Sorin Stratulat

▶ **To cite this version:**

Sorin Stratulat. Integrating Implicit Induction Proofs into Certified Proof Environments. Integrated Formal Methods, 2010, France. pp.320-335. hal-00553070

HAL Id: hal-00553070
https://hal.science/hal-00553070

Submitted on 6 Jan 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Integrating Implicit Induction Proofs into Certified Proof Environments

Sorin Stratulat

LITA, Université Paul Verlaine-Metz, 57000, France
LORIA, 54000, France
stratulat@univ-metz.fr

Abstract. We give evidence of the direct integration and automated checking of implicit induction-based proofs inside certified reasoning environments, as that provided by the Coq proof assistant. This is the first step of a long term project focused on 1) mechanically certifying implicit induction proofs generated by automated provers like Spike, and 2) narrowing the gap between automated and interactive proof techniques inside proof assistants such that multiple induction steps can be executed completely automatically and mutual induction can be treated more conveniently. Contrary to the current approaches of reconstructing implicit induction proofs into scripts based on explicit induction tactics that integrate the usual proof assistants, our checking methodology is simpler and fits better for automation. The underlying implicit induction principles are separated and validated independently from the proof scripts that consist in a bunch of one-to-one translations of implicit induction proof steps. The translated steps can be checked independently, too, so the validation process fits well for parallelisation and for the management of large proof scripts. Moreover, our approach is more general; any kind of implicit induction proof can be considered because the limitations imposed by the proof reconstruction techniques no longer exist. An implementation that integrates automatic translators for generating fully checkable Coq scripts from Spike proofs is reported.

1 Motivations

Implicit induction proof techniques allow for automated reasoning on inductive properties of equational specifications. Up to now, implicit induction theorem provers, as Spike [3], have been successfully used in treating non-trivial case studies, for example, the validation of the JavaCard Platform [3] and the conformance algorithm of a telecommunications protocol [19]. Spike proofs are highly automated; in [3], almost half of the JavaCard bytecode instructions have been checked completely automatically, i.e. do not require users to provide additional lemmas, and done in a reasonable time. These proofs are shallow but large, involving several induction, case analysis and rewriting steps. Even if the theoretical backgrounds of implicit induction proof techniques are widely accepted by the scientific community, their implementation inside theorem provers is error-prone and their certification still stands for a challenge. The most common and

D. Méry and S. Merz (Eds.): IFM 2010, LNCS 6396, pp. 320–335, 2010.
© Springer-Verlag Berlin Heidelberg 2010

ancient validation technique is by human checking. It may work well when the proof size is reasonable, but is not suited for checking many (even easy) inference steps.

To the other extreme, the approach is to certify inference systems such that any proof developed inside certified proof environments is guaranteed to be sound. For example, the proofs done with the Coq proof assistant [24] are mechanically checked by the kernel of its inference system, which is small enough to be human checked and reliable. However, the process for certifying complex software systems is tedious [13], in the case of Spike it would require the validation of thousands of OCaml code lines. The midway approach that we adopted would therefore not consist in certifying inference systems, but in checking proofs that would play the role of test cases for the unreliable systems. More precisely, the Spike proofs are converted into Coq scripts checkable by the Coq kernel. During the last decade, different reasoning tools successfully applied this approach by developing conversion options for Coq [8,4].

The soundness of any implicit induction reasoning can be easily explained using 'proof-by-contradiction' arguments characterizing the 'Descente Infinie' induction-based approaches [21]. For example, proving that a non-empty and potentially infinite set of first-order ground formulas **F** is true requires: 1) (the 'well-foundedness' requirement) a well-founded induction ordering over the formulas from **F**, i.e. there are no infinite strictly descending sequences of formulas, and 2) (the 'counterexample non-minimality' requirement) to prove that for each false formula from **F**, called *counterexample*, there exists a smaller one. The proof starts by assuming by contradiction that there is a counterexample in **F**. Therefore, by 2), there is a smaller counterexample for which is an even smaller counterexample by applying 2) again, and so on. In this way, one can build an infinite strictly descending sequence of counterexamples (hence the name of 'Descente Infinie'), which contradicts 1).

An example of implicit induction proof. Implicit induction proofs as performed by Spike can easily manipulate conjectures about specifications integrating mutually defined functions. Let's consider the universally quantified axioms that mutually define the even, respectively the odd functions over the naturals:

$$even(0) = true \tag{1}$$
$$odd(x) = true \Rightarrow even(S(x)) = true \tag{2}$$
$$odd(x) = false \Rightarrow even(S(x)) = false \tag{3}$$

$$odd(0) = false \tag{4}$$
$$even(x) = true \Rightarrow odd(S(x)) = true \tag{5}$$
$$even(x) = false \Rightarrow odd(S(x)) = false \tag{6}$$

using the constructors 0 and successor S for the naturals. Let's assume that we want to prove the conjectures $odd(S(plus(x,\ x))) = true$ and $even(plus(y,\ y)) = true$, where $plus$ is the addition function over the naturals, defined by the axioms

$plus(0,\ x) = x$ and $plus(S(x),\ y) = S(plus(x,\ y))$. In addition, we assume the lemma $plus(x,\ S(y)) = S(plus(x,\ y))$.

To prove that the conjectures are true (w.r.t the above axioms) using a 'Descente Infinie' induction-based approach means to check the 'well-foundedness' and 'counterexample non-minimality' requirements. For example, the well-founded induction ordering from the first requirement, denoted by $\prec\!\!\prec_{rpo}$, can be defined as a multiset extension of the RPO ordering $\prec_{rpo}$ based on the precedence $0 <_F S <_F plus <_F even$ with multiset status for the defined functions, where *odd* has the same precedence as *even*.[1] The $\prec_{rpo}$ and $\prec\!\!\prec_{rpo}$ orderings can also be used to orient the above axioms from left to right into rewrite rules. In general, the equality $a = b \Rightarrow l = r$ is oriented into $a = b \Rightarrow l \rightarrow r$ if l is the unique greatest term in the equality.

The rewrite rules are involved into rewrite operations that replace terms with smaller ones, essential in the quest for smaller counterexamples during the realization of the 'counterexample non-minimality' requirement. Let's assume that there is a counterexample in the first conjecture, $c_1 : odd(S(plus(n,\ n))) = true$, for some natural n. A smaller counterexample can be pointed out by performing a case analysis on $even(plus(n,\ n))$: if it is $false$ then $odd(S(plus(n,\ n)))$ is $false$ according to (6), i.e. $even(plus(n,\ n)) = false \Rightarrow false = true$; if it is $true$, then we have the tautology $c_3 : even(plus(n,\ n)) = true \Rightarrow true = true$, according to (5). So, $c_4 : even(plus(n,\ n)) = false \Rightarrow false = true$ is a counterexample smaller than $odd(S(plus(n,\ n))) = true$ because we replaced $odd(S(plus(n,\ n)))$ by smaller terms. c_4 can be rewritten with the second conjecture to get the smaller instance $c_2 : even(plus(n,\ n)) = true$. Since the rewritten equality, $c_7 : true = false \Rightarrow false = true$, is a tautology, c_2 is a counterexample.

We can go further and show that there exists at least one counterexample smaller than c_2. Since n is a natural, it can be either 0 or $S(n')$, for some natural n'. If n is 0, $plus(0,\ 0)$ is 0, so $c_6 : even(0) = true$ can be rewritten by (1) to $c_9 : true = true$ which is a tautology. Therefore, n should be $S(n')$. Rewriting $even(plus(S(n'),\ S(n')))$ with the axiom $plus(S(x),\ y) \rightarrow S(plus(x,\ y))$ results $c_5 : even(S(plus(n',\ S(n')))) = true$. By using the lemma $plus(x,\ S(y)) \rightarrow S(plus(x,\ y))$, we obtain the smaller counterexample $c_8 : even(S(S(plus(n',\ n')))) = true$. Another case analysis on $odd(S(plus(n',\ n')))$ yields two new equalities: i) $c_{11} : odd(S(plus(n',\ n'))) = true \Rightarrow true = true$, which is a tautology, and ii) the smaller counterexample $c_{10} : odd(S(plus(n',\ n'))) = false \Rightarrow false = true$. The instance of the first conjecture $c_1' : odd(S(plus(n',\ n'))) = true$ is smaller and can be used to rewrite it into the tautology $c_{12} : true = false \Rightarrow false = true$. Therefore, c_1' is a smaller counterexample. To sum up, c_1' is smaller than the original counterexample c_1. The quest for smaller counterexamples can be similarly repeated *ad infinitum*, which contradicts the 'well-foundedness' requirement. So, we conclude that the conjecture $odd(S(plus(x,\ x))) = true$ is true. For similar reasons, $even(plus(x,\ x)) = true$ is also true.

[1] For more formal definitions, the reader may consult Section 2.

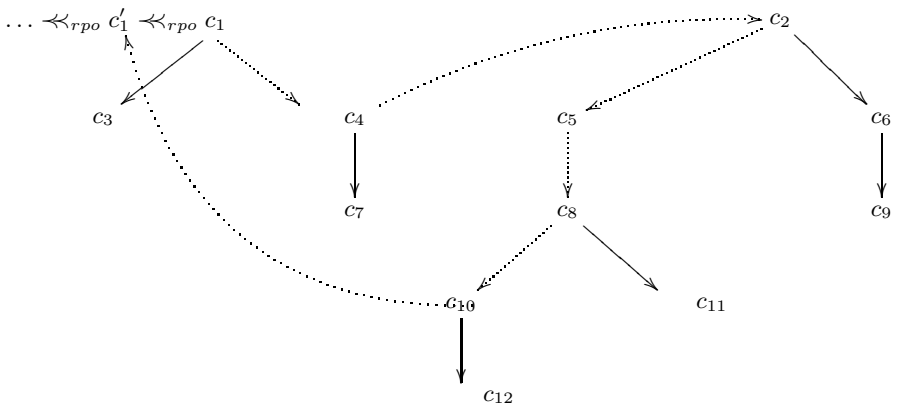

Fig. 1. Example of simultaneous induction proof; the assumption of false initial conjectures generates an infinite strictly descending sequence of counterexamples, as indicated by the dotted arrows

The proof employs *simultaneous induction*, i.e. instances of the first conjecture are used as induction hypotheses in the proof of the second, and viceversa, as depicted by the proof graph from Fig. 1. The nodes are labelled with the names of the ground equalities encountered during the proof. A branching node points out a case analysis operation on the corresponding equality. The solid (resp. dotted) arrows give pathways to tautologies (resp. smaller counterexamples). Notice the infinite pathway of counterexamples which justifies the application of the 'Descente Infinie' induction principle.

Related approaches. Previous attempts to validate Spike proofs have been done by Courant [9] and Kaliszyk [12] using the explicit induction tactics provided by Coq. Their approaches are limited mainly because the explicit induction proof methods require hierarchical manipulation of induction hypotheses; the proofs have a tree-shape such that the exchange of information is forbidden between different branches. Therefore, it is impossible to perform simultaneous induction proofs.

Up to now, the current solution is to reconstruct implicit into explicit induction proofs. In [9], only the proofs done with restricted versions of the Spike system (the K-systems) are considered. Their inference rules have to obey some conditions that would allow proof representations under the form of a tree labelled with judgements. On the other hand, [12] identifies explicit induction schemas from the proof steps that instantiate variables. More recently, Nahon et al. [15] proposed a theoretical foundation based on deduction modulo that permits automated construction of inductive proofs into the sequent calculus, ready for insertion into proof assistants. As Brotherston has already shown in his PhD thesis [6], the idea is to perform 'Descente Infinie'-style proofs using an extension of the sequent calculus with explicit inductive schemas that define

conjectures in terms of induction hypotheses and conclusions linked by shared variables. However, it is difficult to see how the Spike proofs can be reproduced by this calculus since Spike does not assume variable sharing between different conjectures.[2] In our opinion, implicit and explicit induction are just two different proof techniques; they can be even combined during the proof process, as shown in [22].

Some proof assistants already integrate automatizing mechanisms for induction reasoning, for example Coq [25], Agda [14], IsaPlanner [10], NuPrl [16] and Clam [7]. To the best of our knowledge, all of them use explicit inductive definition schemas. On the other hand, there is no similar work w.r.t. the direct integration of implicit induction techniques. The first successful but manual conversion and checking operations of an implicit induction proof by Coq have been reported in [23].

Structure of the paper. The main contribution of the paper is a methodology for mechanically checking potentially any implicit induction proof. We will explain in particular how the methodology works for validating Spike proofs with Coq. After presenting the basic notions and notations in Section 2, we will detail in Section 3 the implicit induction proof techniques, then introduce a simplified version of the Spike inference system but strong enough to prove the introductory example. Section 4 develops the idea of explicitly defining, then implementing the underlying implicit induction principles. The first part will prove the soundness of the 'Descente Infinie' induction principle instantiated for a particular well-founded induction ordering. The second part uses deductive reasoning based on current techniques such as rewriting, case analysis and tautology elimination in order to check the 'counterexample non-minimality' requirement. In addition, we will show and give examples using implementation details that one can build a one-to-one translation of the Spike inference rules into Coq tactics, hence the generality of our approach. The methodology is applied for automatically translating and validating the Spike proof of the introductory example and other non-trivial examples. The conclusions and future work are given in the last section.

2 Basic Notions

Conditional specifications consist of axioms representing conditional equalities between terms built on an alphabet of (arity-fixed) function symbols $\mathcal{F}$ and (universally quantified) variables $\mathcal{V}$. The axioms define some of the function symbols, the other symbols are referred to as constructors. The specifications of interest are many-sorted and we assume that for each sort s there exists at least one constructor of sort s. The conjectures are clauses representing disjunctions of literals, where a literal is either an equality or a inequality between two terms. Sometimes, clauses that have at most one equality are represented as implications.

[2] For more details, the reader may consult the Spike proof of the introductory example from Subsection 3.1.

The set of terms is denoted by $\mathcal{T}(\mathcal{F},\mathcal{V})$ and the set of ground (or no variable) terms by $\mathcal{T}(\mathcal{F})$. New terms (resp. clauses), called *instances*, can be built from existing terms (resp. clauses) by replacing variables with terms. The mappings from variables to terms are called substitutions. Two terms unify if there is a substitution σ such that $s\sigma$ and $t\sigma$ are syntactically equal, denoted by $s\sigma \equiv t\sigma$. A term s matches the term t if there exists a substitution σ such that $s\sigma \equiv t$. The subterm t of a clause C is identified by its position p, denoted by $C[t]_p$.

A *quasi-ordering* $\leq$ is a reflexive and transitive binary relation, consisting of strict and equivalence parts. The strict part of a quasi-ordering is called *ordering* and denoted by $<$. A quasi-ordering $\leq$ defined over the elements of a nonempty set A is *well-founded* if there do not exist infinite strictly descending sequences $\ldots < x_2 < x_1$ of elements of A. A binary relation R is *stable under substitutions* if whenever $s\,R\,t$ then $(s\sigma)\,R\,(t\sigma)$, for any substitution σ. A *reduction* ordering is a transitive and irreflexive relation that is well-founded, stable under substitutions and stable under contexts (i.e. $s\,R\,t$ implies $u[s]\,R\,u[t]$). An example of syntactic reduction ordering over terms is $\prec_{rpo}$ from Section 1. $\prec_{rpo}$ is recursively defined as follows. Given $f \in \mathcal{F}$, a status function τ for $\mathcal{F}$ returns $\tau(f) \in \{lex, mul\}$, foreach $f \in \mathcal{F}$, where *lex* stands for lexicographic status and *mul* for multiset status. Given $<_{\mathcal{F}}$ an ordering over $\mathcal{F}$, an ordering $\prec_{rpo}$ on $\mathcal{T}(\mathcal{F},\mathcal{V})$ is defined as follows: for all terms $s,t \in \mathcal{T}(\mathcal{F},\mathcal{V})$, $t \prec_{rpo} s$ if $s = f(s_1,\ldots,s_m)$ and i) either $s_i = t$ or $t \prec_{rpo} s_i$ for some s_i, $1 \leq i \leq m$, or ii) $t = g(t_1,\ldots,t_n), t_i \prec_{rpo} s$ for all i, $1 \leq i \leq n$ and either a) $g <_{\mathcal{F}} f$, or b) $f = g$ and $(t_1,\ldots,t_n) \prec_{rpo}^{\tau(f)} (s_1,\ldots,s_n)$. $\prec_{rpo}^{lex}$ is the lexicographic extension of $\prec_{rpo}$, i.e. $(a_1,\ldots,a_n) \prec_{rpo}^{lex} (b_1,\ldots,b_n)$ if either i) $a_1 \prec_{rpo} b_1$ or ii) $a_1 = b_1$ and $(a_2,\ldots,a_n) \prec_{rpo}^{lex} (b_2,\ldots,b_n)$. $\prec_{rpo}^{mul}$, also denoted by $\prec\!\!\prec_{rpo}$ in the rest of the paper, is the multiset extension of $\prec_{rpo}$. Two terms s and t are equivalent if either a) $s \equiv t$, or b) $s \equiv f(s_1,\ldots,s_n)$, $t \equiv g(t_1,\ldots,t_n)$, f and g have the same arity and precedence and, for the case when f and g have multiset status, it exists $(t'_1,\ldots,t'_n)$ such that s_i is equivalent with t_i, forall $1 \leq i \leq n$, and $(t'_1,\ldots,t'_n)$ is a permutation of $(t_1,\ldots,t_n)$. The relation $\prec\!\!\prec_{rpo}$ is defined in the next paragraph.

(Conditional) equalities can be transformed into (conditional) rewrite rules of the form $a = b \Rightarrow l \rightarrow r$ if l is greater than a, b and r. A rewrite system $\mathcal{R}$ consists of a set of rewrite rules. The rewrite relation $\rightarrow_{\mathcal{R}}$ denotes rewrite operations only with rewrite rules from $\mathcal{R}$. The reflexive transitive (resp. equivalence) closure of $\rightarrow_{\mathcal{R}}$ is denoted by $\rightarrow_{\mathcal{R}}^{*}$ (resp $\overset{*}{\leftrightarrow}_{\mathcal{R}}$). Given a substitution σ, a rewrite rule $a = b \Rightarrow l \rightarrow r$ and a clause C such that $C[l\sigma]_u$, a *rewrite operation* replaces $C[l\sigma]_u$ by $a\sigma = b\sigma \Rightarrow C[r\sigma]_u$.[3] Any clause can also be represented as the multiset of its literals. A well-founded and stable under substitutions ordering over clauses can be built as the multiset extension of a reduction ordering over terms, as follows. Given two multisets of terms A_1 and A_2, we write $A_1 \prec\!\!\prec_{rpo} A_2$ if, after the pairwise elimination of the equivalent terms from A_1 and A_2, $\forall s \in A_1$, $\exists t \in A_2$ such that $s \prec_{rpo} t$. A_1 and A_2 are equivalent if both of them become empty after the elimination process. Finally, $\Phi_{\prec\!\!\prec_{rpo} C}$ denotes the set $\{\psi\sigma \mid \psi \in$

[3] For details on term rewriting, the reader may consult [1].

Φ, σ a substitution and $\psi\sigma \ll_{rpo} C\}$ of instances of clauses from Φ that are smaller than the clause C.

$s = t$ is an *inductive theorem* of a set of axioms Ax orientable into a rewrite system $\mathcal{R}$ if, for any of its ground instances $s\sigma = t\sigma$, we have $s\sigma \overset{*}{\leftrightarrow}_{\mathcal{R}} t\sigma$. A ground equality is a *counterexample* if it is not an inductive theorem. $s = t$ is *false* if it 'contains' (i.e. one of its instances is) a counterexample. Tautologies are inductive theorems either of the form $(e \Rightarrow)t = t$, where e is an unconditional equality and t a term, or of the form $e_1 \Rightarrow e_2$, where $e_1 \equiv a = b$ and $e_2 \equiv a = b$ (or $b = a$), and a, b are terms.

3 Implicit Induction Proofs with Spike

In the introductory part, we have shown how the 'Descente Infinie' induction principle can help to justify the soundness of implicit induction proofs. We can give a more pragmatic application of this principle since well-founded orderings guarantee the existence of minimal elements. The proof is done by contradiction, by assuming that there is no such minimal element, as follows. We pick an arbitrary element from the set. Since it is not minimal, there exists a smaller one which is not minimal, and so on. In this way, an infinite strictly descending sequence of elements can be built. This contradicts the fact that the ordering is well-founded.

The implicit induction inference systems consist of inference rules that replace a conjecture with a potentially empty set of new conjectures. Proof derivations are built by the successive application of inference rules on an initial set of conjectures. We say that an implicit induction inference system is *sound* if the minimal counterexamples are preserved in the derivations, i.e. whenever an inference rule replaces a conjecture containing a minimal counterexample, there is a further state in the derivation, usually the next state, with a conjecture having an equivalent (w.r.t. well-founded induction quasi-ordering) minimal counterexample. The soundness property is interesting because we can state that the initial set of conjectures are true whenever the derivations end with an empty set of conjectures. Otherwise, assuming that there is a counterexample in the initial set of conjectures, there exists a minimal counterexample in the set of conjectures encountered in the derivation. Since any minimal counterexample is preserved, it should be present in the last state of the derivation. On the other hand, this is not possible because the last state is empty.

From a logical point of view, it is sufficient to show that the replaced minimal counterexample is a consequence of the equivalent minimal counterexample from the further state. The consequence relation is not affected if other true formulas like the axioms and smaller conjectures from the derivation, or equivalent conjectures from further states are involved [20]. These conjectures play the role of *induction hypotheses*.

Implicit induction proofs similar to that presented as example in Section 1 can be highly automated if the quest for smaller or equivalent minimal counterexamples is limited only to the conjectures from the next state. The trick is to store

in the current state previously replaced conjectures that do not contain minimal counterexamples, called *premises*.[4] In this case, the initial set of conjectures are true for any derivation starting with an empty set of premises and finishing with an empty set of conjectures.

3.1 The Spike Prover

Spike is an implicit induction prover that reasons on conditional specifications. In the last decade, it has been successfully used in many real-size case studies from different areas (telecommunications [19], programming language platforms [3], collaborative editing systems [11], web services [18], etc).[5]

Specifications and properties. The conditional specifications are sorted and consist in sets of axioms defining functions, represented as conditional equalities. We assume that a reduction ordering exists such that it can orient the axioms into rewrite rules. The specifications accepted by Spike should be coherent, i.e. any formula and its negation cannot be simultaneously consequences of the axioms, and complete, i.e. the functions are defined in any point of the domain.

A sufficient condition to achieve coherent conditional specifications is the *ground convergence*, i.e. the rewriting process of any ground term terminates and yields a unique result [5]. This property is easier to check for specifications based on free constructors: the lhs of the rewrite rules defining a function symbol f is basic, i.e. of the form $f(\vec{t})$ with $\vec{t} \equiv t_1, \ldots, t_n$ a vector of n constructor terms, and there is no equality relation between two constructors terms starting with different constructor symbols.

Complete specifications may define only *operational sufficiently complete* function symbols f, i.e. for any ground basic term $f(\vec{s})$, i) there are matching axioms $a_i = b_i \Rightarrow l_i \rightarrow r_i$ such that $f(\vec{s}) \equiv l_i\sigma_i$, and ii) $\bigvee_i a_i\sigma_i = b_i\sigma_i$ is an inductive theorem and any two matching substitutions are equivalent modulo renaming. The test for inductive validity is generally undecidable, therefore we will restrict to the case when the specifications contain only conditional axioms with conditions having the form $a = b$, where b is either $true$ or $false$.

The inference system. The Spike inference system is made of inference rules manipulating clauses, representing transitions between states $(E, H) \vdash (E', H')$, where E, E' are conjectures, and H, H' premises. In Fig. 2, we introduce a simplified version of it consisting only of 4 inference rules.

Generate applies on clauses having subterms that unify with some lhs of the axioms, referred to as unifying axioms. The soundness of the system is preserved if all the unifying axioms are considered [3]. Total Case Rewriting can apply on clauses with subterms that are matched by some lhs of conditional axioms. If the position of the subterm resides inside a maximal term of the treated clause C, then C cannot have minimal counterexamples and is added to the set

[4] Notice that the two notions of induction hypothesis and premise are different.

[5] For a more detailed list of publications, see [17].

GENERATE: $(E \cup \{C[t]_p\},\ H) \vdash (E \cup (\cup_\sigma E_\sigma),\ H)$
if E_σ is $\{a\sigma = b\sigma \Rightarrow C\sigma[r\sigma]_p\}$ and $a = b \Rightarrow l \rightarrow r \in Ax$ s.t. $t\sigma \equiv l\sigma$.

TOTAL CASE REWRITING: $(E \cup \{C[t]_p\},\ H) \vdash (E \cup E',\ H \cup \{C\})$
if E' is $\{a_1\sigma_1 = true \Rightarrow C[r_1\sigma_1]_p, a_2\sigma_2 = false \Rightarrow C[r_2\sigma_2]_p\}$ and
$a_1 = true \Rightarrow l_1 \rightarrow r_1$, $a_2 = false \Rightarrow l_2 \rightarrow r_2 \in Ax$ s.t. $l_1\sigma_1 \equiv t$ and $l_2\sigma_2 \equiv t$.

(UNCONDITIONAL) REWRITING: $(E \cup \{C\},\ H) \vdash (E \cup \{C'\},\ H)$
if $C \rightarrow_{Ax \cup \mathcal{L} \cup (H \cup E)_{\prec\prec rpo C}} C'$.

TAUTOLOGY: $(E \cup \{C\},\ H) \vdash (E,\ H)$
if C is a tautology.

Fig. 2. A simplified version of the Spike inference system

of premises. REWRITING rewrites clauses with unconditional orientable axioms Ax, lemmas $\mathcal{L}$, and smaller instances of premises and conjectures. TAUTOLOGY deletes tautologies.

Proof example. The above inference system can prove $e_1 : even(plus(x, x)) = true$ and $e_2 : odd(S(plus(y, y))) = true$ using the lemma $plus(x, S(y)) = S(plus(x, y))$ and the $\prec_{rpo}$, $\prec\prec_{rpo}$ orderings from the introductory example. The initial state of the proof is $(\{e_1, e_2\}, \emptyset)$. TOTAL CASE REWRITE (TCR) is applied on $odd(S(plus(y, y)))$ of e_2 with axioms (5) and (6) to yield $e_3 : even(plus(y, y)) = true \Rightarrow true = true$ and $e_4 : even(plus(y, y)) = false \Rightarrow false = true$. From the current state $(\{e_1, e_3, e_4\}, \{e_2\})$, e_3 is deleted by TAUTOLOGY (T). REWRITING (R) simplifies e_4 with the conjecture e_1 to give the tautology $e_5 : true = false \Rightarrow false = true$, which is deleted in the next step. GENERATE (G) can be applied on the remaining conjecture, e_1. The term $plus(x, x)$ is unified with the lhs of the axioms defining $plus$, to yield $e_6 : even(0) = true$ and $e_7 : even(S(plus(z, S(z)))) = true$, where z is a new variable. e_6 is reduced to the tautology $e_8 : true = true$ by REWRITING with axiom (1), then deleted using TAUTOLOGY. e_7 is reduced to $e_9 : even(S(S(plus(z, z)))) = true$ by REWRITING with the lemma. A second TOTAL CASE REWRITING on the term $even(S(S(plus(z, z))))$ of e_9 yields $e_{10} : odd(S(plus(z, z))) = true \Rightarrow true = true$ and $e_{11} : odd(S(plus(z, z))) = false \Rightarrow false = true$. From the new current proof state $(\{e_{10}, e_{11}\}, \{e_2, e_9\})$, e_{10} is deleted by TAUTOLOGY and e_{11} is reduced to the tautology $e_{12} : true = false \Rightarrow false = true$ by REWRITING with the premise e_2. The proof finishes after the application of TAUTOLOGY on the last conjecture e_{12}.

The proof is schematised as follows: $(\{e_1, \underline{e_2}\}, \emptyset) \vdash^{(\mathrm{TCR})} (\{e_1, \underline{e_3}, e_4\}, \{e_2\}) \vdash^{(\mathrm{T})} (\{e_1, \underline{e_4}\}, \{e_2\}) \vdash^{(\mathrm{R})} (\{e_1, \underline{e_5}\}, \{e_2\}) \vdash^{(\mathrm{T})} (\{\underline{e_1}\}, \{e_2\}) \vdash^{(\mathrm{G})} (\{\underline{e_6}, e_7\}, \{e_2\}) \vdash^{(\mathrm{R})} (\{\underline{e_8}, e_7\}, \{e_2\}) \vdash^{(\mathrm{T})} (\{\underline{e_7}\}, \{e_2\}) \vdash^{(\mathrm{R})} (\{\underline{e_9}\}, \{e_2\}) \vdash^{(\mathrm{TCR})} (\{\underline{e_{10}}, e_{11}\}, \{e_2, e_9\}) \vdash^{(\mathrm{T})} (\{\underline{e_{11}}\}, \{e_2, e_9\}) \vdash^{(\mathrm{R})} (\{\underline{e_{12}}\}, \{e_2, e_9\}) \vdash^{(\mathrm{T})} (\emptyset, \{e_2, e_9\})$. The underlined formula from a proof state is the conjecture to which the corresponding inference rule is applied.

4 Translating and Checking Spike Specifications

Spike specifications and proofs can be directly translated into Coq scripts. Each Spike datatype and function definition is translated into an equivalent Coq representation. In addition, the underlying 'Descente Infinie' induction principle is explicitly defined. In order to do this, Coq formulas have to be syntactically represented and compared as Spike does. The idea is to use a term algebra that abstracts the Coq datatypes and function symbols such that each Coq term is weighted with an abstract term. In this way, comparing two Coq formulas reduces to comparing the weights of their built-in terms.

The Coq scripts consist in the specification and proof parts. The specification part defines the abstract term algebra and the RPO ordering built from abstractions of Spike function symbols and their precedence. Then, the Coq datatypes and functions are introduced, together with translation functions for each datatype, which show how to abstract constructor terms. In the proof part, firstly a weight is associated to each conjecture encountered in the Spike proof, then lemmas about weight comparisons are defined. The 'Descente Infinie' principle is explicitly stated in the main lemma which proves that for each false instance from the set of conjectures, there is another one with a smaller weight. Another lemma states that all conjectures are true. The last lemma trivially concludes that the initial conjectures are true, too.

Datatypes and the function definitions. The Coq datatypes, function definitions and translation functions have to be defined manually by the user. Spike specifications can be annotated with Coq code that is inserted into the generated Coq script during the translation process. As example, here is the code for the introductory example, consisting of a set of computing functions :

```
Fixpoint model_nat (v: nat): term :=
match v with
| O ⇒ (Term id_0 nil)
| S x ⇒ let r := model_nat x in (Term
id_S (r::nil))
 end.
Fixpoint model_bool (v: bool): term
:=
match v with
| true ⇒ (Term id_true nil)
| false ⇒ (Term id_false nil)
end.

Fixpoint plus (x y:nat): nat :=
match x with
| O ⇒ y
| S x' ⇒ S (plus x' y)
end.
```

```
Fixpoint even (v:nat): bool :=
match v with
  | 0 ⇒ true
  | S x ⇒ match odd x with
              | true ⇒ true
              | false ⇒ false
           end
end
with odd (v:nat): bool :=
match v with
  | 0 ⇒ false
  | S x ⇒ match even x with
              | true ⇒ true
              | false ⇒ false
           end
end.
```

where id_x is the abstraction of a function symbol x, model_*sort* is the translation function for *sort* and **term** (recursively defined as `Inductive` **term** : `Set` := | `Var : variable` → **term** | `Term : symbol` → `list` **term** → **term**.) is the type of the abstracted terms provided by COCCINELLE [8], a Coq library well suited for modelling mathematical notions needed for rewriting, such as term algebras and RPO. This is the checking step for the Spike specification and its properties, since any computing function written as a structurally recursive function is guaranteed to be complete and ground convergent, if accepted by Coq. The termination and ground confluence properties result from the fact that every recursive call is executed on a structurally smaller argument and that the inserted Coq script is based on free constructors, respectively.

The ordering over conditional equalities. An important part of the Coq script concerns the implementation of the induction ordering involved in the 'well-foundedness' requirement. Its definition and the computation of comparisons between conjectures are based on computable functions and inductive predicates provided by COCCINELLE. COCCINELLE formalises RPO in a generic way using a precedence and a status (multiset/lexicographic) for each function symbol. Spike automatically generates a term algebra starting from the abstract function symbols which preserve the precedence of the original symbols. Then, the algebra is applied as argument to the functor of the generic RPO module which establishes fundamental properties about RPO orderings, for example, any RPO ordering is a reduction ordering. Also, the well-foundedness of the induction ordering, denoted below by `less`, is provided.

In order to deal with mutually recursive functions, the RPO definition from the generic module has been extended to take into account precedence relations with equivalent symbols. Also, even if many interesting properties about the RPO orderings have been already provided by COCCINELLE, some about the multiset extension of RPO were missing. A new function computing weight comparisons was defined and its equivalence with `less` was proved as a soundness lemma. This guarantees that any terminating weight comparison operation is sound. The termination property is ensured if the size of the terms is limited by a global maximal value. For a given proof, it has to be greater than the double of the maximal size of the terms encountered in the proof. The stability under substitutions of `less` was also proved.

The 'Descente Infinie' induction principle is implemented in three steps. Firstly, the existence of minimal elements in any non-empty set of weights, represented as lists of abstract terms, is guaranteed:

$$\forall\ Y\colon \mathcal{P}\ (\textbf{list term}),\ (\exists\ y,\ y \in\ Y) \rightarrow \exists\ n \in Y,\ \forall m \in\ Y,\ \neg\ (\mathsf{less}\ m\ n).$$

Then, the 'counterexample non-minimality' requirement is implemented such that, for any list of pairs (coq_formula, weight), whenever a formula instance is false there is another one with a smaller weight:

$$\forall F \in \mathbf{F}, \forall\ \overrightarrow{x}, \neg\pi_1(F\ \overrightarrow{x}) \rightarrow (\exists F_1 \in \mathbf{F}, \exists\ \overrightarrow{x_1}, \neg\pi_1(F_1\ \overrightarrow{x_1}) \wedge\ \mathsf{less}\ \pi_2(F_1\ \overrightarrow{x_1})\ \ \pi_2(F\ \overrightarrow{x}))$$

where π_1 and π_2 are the first and second projections of a pair, respectively. **F** is the set of functors that associate a pair (formula, weight) to a vector of terms.

Finally, we prove that $\forall F \in \mathbf{F}, \forall\ \vec{x}, \pi_1(F\ \vec{x})$. The first and the third step require classical reasoning. The third step is valid for any set **F** satisfying the 'counterexample non-minimality' requirement.

Satisfying the 'counterexample non-minimality' requirement. All crucial information for satisfying this requirement can be extracted from the Spike proof, in particular the set **F**, the conjectures containing smaller counterexamples and the comparisons between the weights of the conjecture instances. For example, the set **F** for validating the introductory example consists of functors associated to each of the twelve conjectures e_1 to e_{12}: $\{(fun\ x\ \ =>\ \ (even(plus\ x\ x) = true, weight_{e_1})), \ldots, (fun\ \_\ =>\ (true = false \Rightarrow false = true, weight_{e_{12}}))\}$. The proof consists of a case analysis on the conjectures that may have counterexamples following the 'quest for smaller counterexample' reasoning represented in Fig. 1. Each of the cases can be treated independently and in any order. For example, if e_1 has a counterexample, by performing a case analysis on x instantiated with 0 and $(S\ z)$, a smaller counterexample should exist either in e_6 or e_7.

An important part of the proof is spent on verifying weight comparisons. The comparison proofs can be automatically generated and consist in i) the replacement of all terms of the form (`model_sort` x) with COCCINELLE abstraction variables of the form (`Var` i), where i is a natural, ii) the use of the 'stability under substitutions' property of `less` which allows to perform the comparison tests on weights with abstraction variables instead of using the original weights, iii) computing the comparison result of weights with abstraction variables, and iv) validating the result using the soundness lemma. In order to perform iii) for the case when the weights of two compared terms, $weight_1$ and $weight_2$, consist of m and n abstracted terms, respectively, we have to check that the size of any term from $weight_1$ added with the size of any term from $weight_2$ does not exceed the maximal value. The total number of size comparisons is therefore $m * n$.

One-to-one translation of Spike inference steps. Automatic translators have been implemented for the inference rules from Fig. 2. Deductive steps like rewriting, case analysis and tautology elimination operations are directly translated into Coq proof commands.

The variable instantiation schemas of GENERATE are controlled by Coq functional schemas [2]. This is the checking step for complete instantiation schemas. For example, to show that x from e_1 is replaced by 0 and $(S\ z)$, we define a function f with all the instantiation cases:

```
Fixpoint f (x: nat) {struct x} : nat :=
 match x with
| 0 ⇒ 0
| (S z) ⇒ 0
end.
```

Functional `Scheme` *f_ind* := `Induction for f Sort Prop.`

The instances are generated by the Coq script `pattern` x, (f x). `apply` f_ind. The idea is that, for each instance $HFabs$, we choose the functor from **F** corresponding to the appropriate conjecture from the Spike proof script, and show that $HFabs$ is logically equivalent with a smaller instance. For the case when x is 0, here is the generated Coq script: `exists` (`fun _` $\Rightarrow$ ((even 0) = true, $weight_{e6}$)). `eexists`. `split`. `contradict` $HFabs$. `auto`. `apply` less_HFabs_e6. The first two commands instantiate e_6, then the proof is split in two: the 'logically equivalence' and comparison parts consisting of the application of `auto`, a tactic enough powerful to show the equivalence between `plus` 0 0 and 0, and of the comparison lemma less_HFabs_e6, respectively.

The translation of Total Case Analysis is similar, excepting that the case analysis on whether a condition a is either $true$ or $false$ is performed by `destruct` a. This is the checking step for the inductive validity of the disjunction of rewrite rules conditions. This translation offers a better control of the rewritten term than `auto`. For example, the fact that $C[f(t)]$ is rewritten with a rule of the form $f(x) \rightarrow \ldots$ can be simulated by `pattern` t. `simpl` f. `cbv beta`. `pattern` t isolates t from C, `simpl` f rewrites $f(t)$ and `cbv beta` puts back the resulted term in C. Rewriting is translated using the same trick for rewriting.

There is no need to reduce or compare tautologies with other conjectures. Tautologies can be eliminated by Coq using `intros`. `auto`. `intros` separates the conditions from the conclusion of a conditional equality. `auto` checks either that the conclusion is of the form $t = t$ or that the conditions contain the conclusion.

Experimental results. Table 1 displays some statistics about the execution time and size of the Coq scripts generated with our implementation for several Spike specifications and conjectures: properties about *plus* and different definitions of *even* and *odd* (conjectures 1] to 10]), about other recursive data structures like trees and lists (conjectures 11] and 12]) and conjectures stating the soundness of a simple insertion sorting algorithm (conjectures 13] to 16]).

The third and fourth columns show the number of comparison lemmas and the time needed for their validation, respectively. The fifth and sixth columns display the cardinality of **F** and the validation time of the corresponding formulas, respectively. The last column gives the total execution time, including the overhead time needed for checking the algebra and the ordering. The overhead time for the first ten conjectures was 45.5s, for the next two conjectures was 1m10s and for the last conjectures 59.6s. The statistics for 5], 6] and 7] take into account that 4] was executed before being used as lemma. On the other hand, 16] requires a lemma whose proof in Spike needs arithmetic reasoning. In Coq, the lemma was represented as a hypothesis. Finally, the execution time of each of the Spike proofs and the translation operations lasted less than one second.

The experiments have been done on a MacBook Air featuring a 1.6 GHz Intel Core 2 Duo processor and 2 GB RAM.

Table 1. Statistics about the Coq validation process of some implicit induction proofs

#	conjecture(s)	# less	less time	# **F**	proof time	total time
1]	$evenr(plus(x,y)) = true \wedge$ $evenr(plus(y,z)) = true$ $\Rightarrow evenr(plus(x,z)) = true$	27	3m15s	22	0m50.5s	4m51s
2]	$evenm(x) = evenr(x)$	9	0m10s	8	0m00.5s	0m56s
3]	$plus(x,0) = x$	4	0m02s	3	0m02s	0m48s
4]	$plus(x,S(y)) = S(plus(x,y))$	7	0m01.5s	8	0m01.5s	0m56s
5]	$even(plus(x,x)) = true$ and $odd(S(plus(x,x))) = true$ (needs 4])	17	0m44s	20	0m06.5s	1m34s
6]	$plus(x,y) = plus(y,x)$ (needs 4])	26	0m39s	24	0m05.5s	1m30s
7]	$evenm(plus(x,x)) = true$ and $oddm(plus(x,x)) = false$ (needs 4])	21	1m13s	20	0m13.5s	2m12s
8]	$evenr(S(x)) = true \Rightarrow$ $true = oddm(x)$	11	0m49s	10	0m10.5s	1m45s
9]	$oddc(x) = oddm(x)$	19	1m09s	16	0m29.5s	2m24s
10]	$plus(x,plus(y,z)) = plus(plus(x,y),z)$	7	0m46s	6	0m23.5s	1m55s
11]	$flat(ins(x,t)) = Cons(x,flat(t))$	14	0m53s	15	0m38.4s	2m21s
12]	$app(x,app(y,z)) = app(app(x,y),z)$	8	0m46s	17	0m22.4s	1m58s
13]	$sorted(Cons(x,y)) = true \Rightarrow$ $sorted(y) = true$	5	0m07s	6	0m30s	1m47s
14]	$length(insert(x,y)) = S(length(y))$	9	0m28s	10	0m27s	2m05s
15]	$length(isort(x)) = length(x)$	14	0m36s	16	0m32s	2m18s
16]	$sorted(isort(x)) = true$ (needs lemma)	5	0m02s	4	0m27s	1m39s

5 Conclusions and Future Work

We have proposed a methodology for directly checking potentially any implicit induction proofs using certified proof environments. By the means of the Coq proof assistant, the methodology was applied to check non-trivial proofs done with a restricted version of the Spike system. The 'Descente Infinie' induction principle underlying the Spike proofs was explicitly defined and every single Spike inference step has been translated into equivalent Coq script using automated translators.

One of our long-term goals is to automatically check large Spike proofs. As shown by the experimental results, the checking time is some orders of magnitude longer than for producing a Spike proof, so the current implementation has to be optimised. To meet this objective, the fixed part of the scripts (i.e. the specification and the RPO ordering definition) can be validated in a separate Coq module to be imported, instead of being (re)validated each time a new conjecture is proved. Also, a lot of time is spent validating comparison lemmas. Computing the size of all terms in advance would linearize the complexity of comparison proofs. Last but not least, since the translated inference steps can be performed independently, it would be interesting to check them concurrently.

Some other Spike proofs require more sophisticated inference rules to deal with arithmetic reasoning, parametrized specifications and existential variables [3], or more general versions of the presented inference rules, for example TOTAL CASE REWRITING with conditional axioms having more complex conditions. A challenge would be to implement automatic translators for each of these cases.

In other direction, we intend to define a tactic that performs implicit induction reasoning as an alternative to the existing explicit induction techniques for validating inductive properties. In this way, Coq (and other similar proof assistants) would be able to automatically execute multiple induction steps and manage more conveniently mutually defined functions.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
2. Barthe, G., Courtieu, P.: Efficient reasoning about executable specifications in Coq. In: Theorem Proving in Higher Order Logics, p. 64 (2002)
3. Barthe, G., Stratulat, S.: Validation of the JavaCard platform with implicit induction techniques. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 337–351. Springer, Heidelberg (2003)
4. Bonichon, R., Delahaye, D., Doligez, D.: Zenon: An extensible automated theorem prover producing checkable proofs. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 151–165. Springer, Heidelberg (2007)
5. Bouhoula, A., Rusinowitch, M.: Implicit induction in conditional theories. Journal of Automated Reasoning 14(2), 189–235 (1995)
6. Brotherston, J.: Sequent Calculus Proof Systems for Inductive Definitions. PhD thesis, University of Edinburgh (November 2006)
7. Bundy, A., van Harmelen, F., Horn, C., Smaill, A.: The Oyster-Clam system. In: CADE-10, pp. 647–648. Springer, Heidelberg (1990)
8. Contejean, E., Courtieu, P., Forest, J., Pons, O., Urbain, X.: Certification of automated termination proofs. In: Frontiers of Combining Systems, pp. 148–162 (2007)
9. Courant, J.: Proof reconstruction. Research Report RR96-26, LIP, Preliminary version (1996)
10. Dixon, L.: A Proof Planning Framework for Isabelle. PhD thesis, University of Edinburgh (2005)
11. Imine, A., Rusinowitch, M., Oster, G., Molli, P.: Formal design and verification of operational transformation algorithms for copies convergence. Theoretical Computer Science 351(2), 167–183 (2006)
12. Kaliszyk, C.: Validation des preuves par récurrence implicite avec des outils basés sur le calcul des constructions inductives. Master's thesis, Université Paul Verlaine - Metz (2005)
13. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an operating-system kernel. Communications of the ACM 53(6), 107–115 (2010)
14. Lindblad, F., Benke, M.: A tool for automated theorem proving in Agda. In: TYPES, pp. 154–169 (2004)
15. Nahon, F., Kirchner, C., Kirchner, H., Brauner, P.: Inductive proof search modulo. Annals of Mathematics and Artificial Intelligence 55(1-2), 123–154 (2009)

16. Pientka, B., Kreitz, C.: Automating inductive specification proofs in NuPrL. Fundamenta Informaticae 1(2), 182–209 (1998)
17. The Spike prover, `http://code.google.com/p/spike-prover`
18. Rouached, M., Godart, C.: Reasoning about events to specify authorization policies for web services composition. In: ICWS, IEEE International Conference on Web Services, pp. 481–488. IEEE Computer Society, Los Alamitos (2007)
19. Rusinowitch, M., Stratulat, S., Klay, F.: Mechanical verification of an ideal incremental ABR conformance algorithm. J. Autom. Reasoning 30(2), 53–177 (2003)
20. Stratulat, S.: A general framework to build contextual cover set induction provers. J. Symb. Comput. 32(4), 403–445 (2001)
21. Stratulat, S.: Automatic 'Descente Infinie' induction reasoning. In: Beckert, B. (ed.) TABLEAUX 2005. LNCS (LNAI), vol. 3702, pp. 262–276. Springer, Heidelberg (2005)
22. Stratulat, S.: Combining rewriting with Noetherian induction to reason on non-orientable equalities. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 351–365. Springer, Heidelberg (2008)
23. Stratulat, S., Demange, V.: Validating implicit induction proofs using certified proof environments. In: Poster Session of 2010 Grande Region Security and Reliability Day, Saarbrucken (March 2010)
24. The Coq Development Team. The Coq reference manual - version 8.2 (2009), `http://coq.inria.fr/doc`
25. Wilson, S., Fleuriot, J., Smaill, A.: Inductive proof automation for Coq. In: Coq Workshop (to appear 2010)