Mechanically Certifying Formula-based Noetherian Induction Reasoning

Sorin Stratulat

Mechanically Certifying Formula-based Noetherian Induction Reasoning

Sorin Stratulat

Université de Lorraine
LITA, Department of Computer Science
Ile du Saulcy, Metz, 57000, FRANCE

Abstract

In first-order logic, the formula-based instances of the Noetherian induction principle allow to perform effectively simultaneous, mutual and lazy induction reasoning. Compared to the term-based Noetherian induction instances, they are not directly supported by the current proof assistants.

We provide general formal tools for certifying formula-based Noetherian induction proofs by the Coq proof assistant, then show how to apply them to certify proofs of conjectures about conditional specifications, built with: i) a reductive rewrite-based induction system, and ii) a reductive-free cyclic induction system. The generation of reductive proofs and their certification process can be easily automatised, *without* requiring additional definitions or proof transformations, but may involve many ordering constraints to be checked during the certification process. On the other hand, the reductive-free proofs generate fewer ordering constraints, may involve more general specifications and the certification process is more effective. However, their proof generation is less automatic and the generated proofs need to be normalised before being certified. The methodology for certifying reductive-free cyclic induction proofs related to conditional specifications extends a previous approach used for implicit induction proofs and it can be easily adapted to certify *any* formula-based Noetherian induction reasoning.

In practice, the methodology has been implemented to automatically certify implicit induction proofs generated by the SPIKE theorem prover as well as reductive-free cyclic proofs built by the same system but in a less automatic way.

Key words: Noetherian induction, cyclic induction proofs, implicit induction proofs, proof certification, Coq, SPIKE

Email address: sorin.stratulat@univ-lorraine.fr (Sorin Stratulat).

1. Motivations

Formal verification of mathematical reasoning has recently witnessed remarkable breakthroughs, as revealed by the Flyspeck project (The Flyspeck website, 2014) and the certified proof of the Feit-Thompson theorem (Gonthier et al., 2013). Similar advances can be noticed for the complete validation of complex and crucial systems from the everyday life of computer scientists like compilers (The CompCert project, 2014) and kernels of operating systems (The L4.verified project, 2014).

Induction reasoning plays a central role among the mathematical techniques employed in formal verification as it is well-adapted to reason on recursive data structures and algorithms. *Noetherian* induction is a general reasoning principle for checking whether a property is satisfied by an unbounded set $\mathcal{E}$ of partially ordered elements. When checking the property for some element, the principle grants the fact that the property holds for smaller elements if the (induction) ordering relation is *well-founded* (or Noetherian), i.e., every non-empty subset of $\mathcal{E}$ has a minimal element. The checked property is referred to as *induction conclusion* and the granted properties as *induction hypotheses* (IHs). When applied to first-order reasoning, the elements can be (vector of) terms or (first-order) formulas.

The term-based Noetherian induction methods build *explicit* induction schemas attaching *eagerly* IHs to induction conclusions. In this approach, the proof of an induction conclusion is further developed, expecting that the IHs be applied at some proof step. Defining the right induction schemas may need several proof attempts, especially when some knowledge about the way the proof will be performed is lacking. In practice, it may happen that IHs be defined but not used or that IHs be required but not defined. The latter case is challenging especially when defining induction schemas for the management of mutual induction reasoning where a property can help proving another property, and conversely.

A major advantage for using explicit induction schemas is their direct implementation into inference systems producing tree-shaped proofs by the means of *induction rules*. The nodes of the proof tree are tagged with formulas, the induction reasoning being applied locally, at the node level. Explicit induction schemas can be automatically generated from the analysis of recursive function or datatype definitions, a feature that is implemented by many modern formal reasoning tools like the Coq proof assistant (The Coq development team, 2013).

On the other hand, the formula-based Noetherian induction methods can perform *lazy* induction, a very useful feature that provides the IHs by need, mutual induction as well as *simultaneous* induction where several induction steps are performed at the same time and on different conjectures. However, these methods are hard to be applied at node level in tree-shaped proofs because the formula-based Noetherian induction reasoning may involve information from potentially any node of the proof. The lazy IHs should be eventually discharged independently from the induction conclusion, which makes impossible the mutual induction reasoning.

The formula-based induction ordering is unique during any proof session developed by *reductive* induction techniques like the rewrite-based *implicit induction*, suggested in (Kounalis and Rusinowitch, 1990) and defined later in (Bronsard et al., 1994). In this case, the induction ordering can guide the proof such that the new formulas issued after

processing any formula ϕ are smaller than (and sometimes equal to) ϕ or instances of ϕ. In general, the reductive induction reasoning can be easily automatised and multiple induction steps can be performed during a proof session, as it is witnessed by the proofs generated with the implicit induction SPIKE prover (The SPIKE development team, 2014).

The non-trivial Noetherian induction reasoning for first-order logic is *cyclic* in the sense that the proof of a formula ϕ depends, directly or indirectly, on (an instance of) ϕ. Compared to other induction techniques, the cyclic induction methods, as those using semantic orderings to reason on first-order logic with inductive definitions (FOL_{ID}) (Brotherston and Simpson, 2011) or based on well-founded orderings (Stratulat, 2012), put in evidence cycles of formulas whose proofs are mutually dependent. The soundness of the non-trivial induction reasoning is established if some ordering constraints defined over the formulas inside a cycle are satisfied. Their number may differ, according to the employed induction technique. In (Stratulat, 2012), it has been shown to be equal to i) 1, for cycles built using term-based Noetherian induction methods, ii) to the number of formulas from a cycle, for cycles using reductive formula-based Noetherian induction techniques, and iii) to the (much smaller) number of formulas from a cycle whose instances are used as IHs by other formulas from the cycle, for cycles built with reductive-free formula-based Noetherian induction methods.

The manual check of the sound application of the formula-based principles may be tedious and error-prone. For these reasons, the mechanical certification of formula-based Noetherian induction reasoning becomes a necessity. To our knowledge, there is no formal system that can *directly* certify formula-based Noetherian induction reasoning. Some certifying proof environments based on type theory, such as Coq and Isabelle (Nipkow et al., 2002), have a higher-order specification language that allow the definition and application of the Noetherian induction principle but only the term-based Noetherian induction is supported at first-order level by providing the formal tools to build explicit induction schemas.

In Coq, the recursion-based specifications can be formalised in two ways, by the means of: i) the *functional programming* style, based on pattern matching constructions, and ii) the *logic programming* style, based on inductively defined predicates. Both styles allow to encode recursion but each of them has limitations. Any function should be total and terminating, the built-in termination criterion being supported by a structural recursion analysis checking that one of the function arguments decreases according to a well-founded ordering. On the other hand, any inductive predicate is defined by the means of a set of formulas written in a 'Horn-clause' implication form, no termination proof is needed but no induction principle can be derived from its definition.

It is important to notice that, from a theoretical point of view, the term-based Noetherian induction principle can be trivially represented as an instance of the formula-based Noetherian induction principle (Stratulat, 2012). In the other direction, it is not clear that any formula-based Noetherian induction proof can be converted to a term-based Noetherian induction proof. For example, in the setting of FOL_{ID}, it is conjectured in (Brotherston and Simpson, 2011) that the cyclic proofs can be converted to explicit induction proofs.

Contributions As an alternative way to the use of explicit induction proof methods in Coq, we point out the possibility to certify *any* formula-based Noetherian induction reasoning *as it is*, i.e., with no proof transformation techniques for conversion to term-based Noetherian induction reasoning. In order to do this, we show how to build formal tools for defining the formula-based Noetherian induction principle and the underlying induction orderings. As a case study, we have focussed on proving properties about conditional specifications. Two proof techniques, implementing the implicit and cyclic induction, are presented. The implicit induction method applies only once the formula-based Noetherian induction principle on the set of *all* formulas from the proof (modulo some redundancy criteria). On the other hand, the cyclic method based on well-founded orderings (Stratulat, 2012) may apply the formula-based Noetherian induction principle several times and is more flexible by allowing the use of different induction orderings in a proof session. The certification process is more effective since it deals with smaller sets of formulas and fewer ordering constraints to be checked.

Based on these formal tools, we propose a methodology for certifying formula-based Noetherian induction proofs by translating them into Coq script that can be validated by the Coq's trustworthy kernel. The practical interest of our approach is witnessed by the Coq certification of i) a simultaneous induction proof of the 'P and Q' example (Wirth, 2004), and ii) a non-trivial implicit induction proof of a property over the *even* and *odd* function symbols defined over naturals and requiring several induction steps. The implicit induction proof was automatically generated by SPIKE and automatically translated in Coq script using the functional programming style. The proof of the 'P and Q' example was generated by hand, using a toy and simulating a not-yet implemented reductive inference system, and certified by Coq using a logic programming style. However, the certification methodology is general and we expect it to help implementing in Coq other formula-based Noetherian induction methods, as the saturation-based inductionless induction method (Comon, 2001) (also known as proof by consistency (Kapur and Musser, 1987)). It also opens the perspective of directly integrating the formula-based Noetherian induction reasoning in Coq and similar systems.

Related works Since not all formula-based Noetherian induction proofs are directly representable as term-based Noetherian induction proofs, some effort has been put into finding classes of convertible formula-based Noetherian induction proofs or defining the appropriate translation methods. Related to the Coq development, (Courant, 1996; Kaliszyk, 2005) proposed partial solutions for converting implicit to explicit induction proofs. Courant (Courant, 1996) identified a class of implicit induction inference systems that can generate such convertible proofs. The downside of this approach is the lack of certification for the conversion process, required to fully certify the implicit induction proofs. Deplagne, C. Kirchner, H. Kirchner and Nguyen (Deplagne et al., 2003) established sufficient conditions for identifying the implicit induction proofs that can be directly represented as term-based Noetherian induction proofs, the induction reasoning being embedded in a deduction modulo inference system.

In general, lazy and mutual induction reasoning can also be performed with schemata-based induction, but in a limited way. Protzen (Protzen, 1994) proposed a solution to generate lazily the IHs by building the induction schemas during the proof using proof analysis and not the usual recursion analysis of data structures. On the practical side, Voicu and Li (Voicu and Li, 2009) implemented a Coq tactic that automatically does lazy

term-based 'Descente Infinie' induction reasoning, a contrapositive form of the Noetherian induction reasoning. Also, the mutual induction reasoning may be tricky since it consists in the definition of induction schemas issued from the analysis of classes of mutually defined predicates/functions or data structures (Boyer and Moore, 1988; Walther, 1993; Kapur and Subramaniam, 1996; Boulton and Slind, 2000) and it is not clear that all the necessary IHs can be generated from the application of such induction schemas (Liu and Chang, 1987).

In Coq, induction principles can also be generated from the analysis of function definitions, using the Function (Balaa and Bertot, 2000; Barthe and Courtieu, 2002; Barthe et al., 2006) Equations (Sozeau, 2010) and Program (Sozeau, 2007) tools. These principles are powerful as they can handle dependent pattern-matching and analyse recursion for higher-order type theory. But, once restricted to first-order logic, they implement the term-based Noetherian induction principle.

Stratulat and Demange (Stratulat, 2010; Stratulat and Demange, 2011) succeeded to formalize the implicit induction reasoning directly into Coq scripts. Their goal was to automatically certify implicit induction proofs generated by native implicit induction provers as SPIKE. In this approach, the specifications and proofs are automatically translated into Coq scripts. At the end, the generated scripts are checked for conformity with the formalised formula-based Noetherian induction principle. However, this approach does not certify the translation process between the SPIKE proofs and Coq scripts. It has been shown in (Stratulat and Demange, 2011) that non-trivial conjectures can be successfully certified in an automatic way. In another direction, Henaien and Stratulat (Henaien and Stratulat, 2013) developed Coq tactics that can prove goals by implicit induction using SPIKE as an external tool. They firstly translate in SPIKE the Coq specifications and goals, then call SPIKE to develop the implicit induction proofs of the goals. Based on the previous approach, the SPIKE proofs are further converted to Coq scripts which are finally certified as proofs of the original Coq goals.

Structure of the paper The paper is organised in 7 sections and one appendix, as follows. The basic notions and notations are introduced in Section 2. The Coq formalisation of the formula-based Noetherian induction principle and the certification methodology are presented in Section 3. Section 4 introduces different implicit induction inference systems. A first attempt to certify the proof of the 'P and Q' example is presented, then the certification of the implicit proof of the property on *even* and *odd* is explained using a functional programming style. The proofs of these properties are redone in Section 5 using cyclic inference systems and showed that the certification of the cyclic proof concerning the 'P and Q' example is successful by using a logic programming style. The SPIKE system, the instructions for generating implicit and cyclic proofs as well as their conversion into Coq script are presented in Section 6. Section 7 concludes and outlines future work. Related Coq and SPIKE scripts are included in the appendix. [1]

[1] The full source code, including the used libraries, reasoning systems, specifications and proofs, is provided as supplementary material at `http:/lita.univ-lorraine.fr/~stratula/jsc-pas.zip` and on SPIKE's website `https://github.com/sorinica/spike-prover`.

2. Basic notions

Syntax We consider an alphabet of arity-fixed function symbols $\mathcal{F}$ and predicate symbols $\mathcal{P}$. We also consider an enumerable set of variables $\mathcal{V}$ and denote by $\mathcal{T}(\mathcal{F},\mathcal{V})$ the set of terms built on function symbols from $\mathcal{F}$ and variables from $\mathcal{V}$. The expression $P(\overline{t})$ is an *atom*, where $P \in \mathcal{P}$ is an n-ary predicate and $\overline{t}$ is the *vector of terms* $(t_1,\ldots,t_n)$ such that $t_i \in \mathcal{T}(\mathcal{F},\mathcal{V})$, for any $i \in [1..n]$. The set $\mathcal{A}(\mathcal{P},\mathcal{F},\mathcal{V})$ denotes the set of atoms over $\mathcal{P}$, $\mathcal{F}$ and $\mathcal{V}$. $\mathcal{T}(\mathcal{F})$ (resp., $\mathcal{A}(\mathcal{P},\mathcal{F})$) is a shortcut for $\mathcal{T}(\mathcal{F},\emptyset)$ (resp., $\mathcal{A}(\mathcal{P},\mathcal{F},\emptyset)$) and represents the set of ground terms (resp., ground atoms). In the following, we will consider only quantifier-free first-order formulas over $\mathcal{A}(\mathcal{P},\mathcal{F},\mathcal{V})$.

Given a term or formula ϕ, we denote by $Var(\phi)$ the set of variables from ϕ. *Instances* of ϕ are resulted by replacing some of its variables by terms, by the means of *substitutions*. Any substitution is a finite set of n (> 0) mappings $\bigcup_{i=1}^{n}\{x_i \mapsto t_i\}$, also denoted as $\{\overline{x} \mapsto \overline{t}\}$, where $\overline{x} \equiv (x_1,\ldots,x_n)$, $\overline{t} \equiv (t_1,\ldots,t_n)$, $x_i \in \mathcal{V}$, $t_i \in \mathcal{T}(\mathcal{F},\mathcal{V})$, for each $i \in [1..n]$, and $\equiv$ denotes the syntactic equality. The instance of a term t built with the substitution σ is denoted by $t\sigma$. σ^t_{id} denotes the *identity* substitution $\{x \mapsto x \mid x \in Var(t)\}$. Similarly, the notion of instance can be extended for atoms, formulas and (multi)sets of formulas. A term or formula ϕ that includes another term s is denoted by $\phi\langle s\rangle$. A *ground substitution* has ground all its mapping terms.

Orderings A *quasi-ordering* $\leq$ is a reflexive and transitive binary relation. Its strict part, referred to as *ordering*, is denoted by $<$ and its equivalent part by $\sim$. A quasi-ordering defined over the elements of a nonempty set $\mathcal{E}$ is *well-founded* if and only if every non-empty subset of $\mathcal{E}$ has a minimal element. Under some choice assumption, it is equivalent to asking that any strictly decreasing sequence of elements of $\mathcal{E}$ is finite. A binary relation R is *stable under substitutions* if whenever $s\,R\,t$ then $(s\sigma)\,R\,(t\sigma)$, for any substitution σ and terms or formulas s and t. R is *stable under contexts* if, for any two terms s and t such that $s\,R\,t$ and any term $l\langle s\rangle$, then $l\,R\,r$, where r can be any term built by replacing in l occurrences of s by t. A quasi-ordering is stable under substitutions if its strict and equivalent parts are stable under substitutions. A *reduction* ordering is a transitive and irreflexive relation that is well-founded and stable under substitutions and contexts. Given a set of formulas Ψ, by $\Psi_{\leq\psi}$ (resp., $\Psi_{<\psi}$) are denoted the instances of formulas from Ψ that are smaller or equal (resp., strictly smaller) than ψ w.r.t. $\leq$ (resp., $<$).

An example of syntactic reduction ordering over terms is the *recursive path ordering* (rpo) (Dershowitz, 1982; Kamin and Lévy, 1980; Lescanne, 1983). Let us assume the *status* function τ for $\mathcal{F}$ that returns $\tau(f) \in \{\mathsf{Lex},\mathsf{Mul}\}$, foreach $f \in \mathcal{F}$, where Lex (resp., Mul) stands for lexicographic (resp., multiset) status. Given a precedence over $\mathcal{F}$, denoted by the well-founded quasi-ordering $\leq_{\mathcal{F}}$, the rpo $\prec_{rpo}$ is recursively defined, as follows: for all terms $s,t \in \mathcal{T}(\mathcal{F},\mathcal{V})$, $t \prec_{rpo} s$ if $s \equiv f(s_1,\ldots,s_m)$ and i) either $s_i \equiv t$ or $t \prec_{rpo} s_i$ for some s_i, $1 \leq i \leq m$, or ii) $t \equiv g(t_1,\ldots,t_n), t_i \prec_{rpo} s$ for all i, $1 \leq i \leq n$ and either a) $g <_{\mathcal{F}} f$, or b) $f \sim_{\mathcal{F}} g$, f and g have the same arity and status, and $(t_1,\ldots,t_n) \prec_{rpo}^{\tau(f)} (s_1,\ldots,s_n)$. $\prec_{rpo}^{\mathsf{Lex}}$ is the *lexicographic extension* of $\prec_{rpo}$: $(a_1,\ldots,a_n) \prec_{rpo}^{\mathsf{Lex}} (b_1,\ldots,b_n)$ if either i) $a_1 \prec_{rpo} b_1$ or ii) $a_1 \sim_{rpo} b_1$ and $(a_2,\ldots,a_n) \prec_{rpo}^{\mathsf{Lex}} (b_2,\ldots,b_n)$, where $\sim_{rpo}$ is recursively defined as: $t \sim_{rpo} t$, for any term t, and $f(a_1,\ldots,a_n) \sim_{rpo} g(b_1,\ldots,b_n)$ if $f \sim_{\mathcal{F}} g$ and, for each $i \in [1..n]$, $a_i \sim_{rpo} b_i$. Two terms s and t are *equivalent* if $s \sim_{rpo} t$.

$\prec_{rpo}^{\mathsf{Mul}}$, also denoted by $\prec\!\!\prec_{rpo}$, is the *multiset extension* of $\prec_{rpo}$: $(A \equiv\ )(a_1, \ldots, a_n) \prec\!\!\prec_{rpo} (b_1, \ldots, b_n)(\ \equiv B)$ if for any term $a \in A'$, there is a term $b \in B'$ such that $a \prec_{rpo} b$, where the multiset A' (resp., B') collects the terms from A (resp., B) after deleting pairwisely the equivalent terms from A and B. Two multisets of terms are equivalent if they are reduced to empty sets after deleting pairwisely their equivalent terms. The multiset extension of any well-founded ordering is also well-founded (Baader and Nipkow, 1998).

Noetherian induction principles for first-order logic In its most general form, the Noetherian induction principle is formalised as

$$(\forall m \in \mathcal{E}, (\forall k \in \mathcal{E}, k < m \Rightarrow \phi(k)) \Rightarrow \phi(m)) \Rightarrow \forall p \in \mathcal{E}, \phi(p)$$

where $(\mathcal{E}, \leq)$ is a poset, $<$ a well-founded ordering and ϕ the property to be checked for any element of $\mathcal{E}$. The formulas $\phi(k)$ are called *induction hypotheses* (IHs) and $\phi(m)$ *induction conclusion*.

When $\mathcal{E}$ consists of an *unbounded* number of elements, the set of formulas to be proved and the induction reasoning should be *finitely* represented, by means of variables and *induction schemas*. For example, let us assume that $\mathcal{E}$ is the set $\mathbb{N}$ of natural numbers, ϕ is a unary predicate taking naturals as argument, and $<$ is the 'less than' ordering relation over naturals. By Noetherian induction, the infinite set of conjectures $\{\phi(0), \phi(s(0)), \phi(s(s(0))), \ldots\}$ is entailed from the set of weaker formulas $\{\phi(0), \phi(0) \Rightarrow \phi(s(0)), \phi(s(0)) \Rightarrow \phi(s(s(0))), \ldots\}$. This can be factorised and finitely represented under the form of the Peano principle: $(\phi(0) \wedge \forall x \in \mathbb{N}, \phi(x) \Rightarrow \phi(s(x))) \Rightarrow \forall x \in \mathbb{N}, \phi(x)$, where 0 and s (the 'successor' function) are the constructor symbols for naturals and the variable x from $\forall x \in \mathbb{N}, \phi(x)$ an *induction variable*. An *induction schema* is a finite description of the set of weaker formulas. It is built from IH-free formulas, called *base* cases, and IH-based formulas, called *step* cases. For instance, the induction schema underlying the Peano principle consists of one base case ($\phi(0)$) and one step case ($\forall x \in \mathbb{N}, \phi(x) \Rightarrow \phi(s(x))$).

Restricted to *first-order* reasoning, the Noetherian induction principle can be instantiated in two distinct ways (Stratulat, 2012), whether the elements of $\mathcal{E}$ are (vectors of) terms or (first-order) formulas:

- *term-based* Noetherian induction: ($\forall$ term vector $\overline{m} \in \mathcal{E}$, ($\forall$ term vector $\overline{k} \in \mathcal{E}, \overline{k} <_t \overline{m} \Rightarrow \phi(\overline{k})) \Rightarrow \phi(\overline{m})) \Rightarrow \forall$ term vector $\overline{p} \in \mathcal{E}, \phi(\overline{p})$.
- *formula-based* Noetherian induction: ($\forall$ formula $\delta \in \mathcal{E}$, ($\forall$ formula $\gamma \in \mathcal{E}, \gamma <_f \delta \Rightarrow \phi(\gamma)) \Rightarrow \phi(\delta)) \Rightarrow \forall$ formula $\rho \in \mathcal{E}$, $\phi(\rho)$. Let us notice that ϕ is a second-order predicate. In order to downgrade to first-order reasoning, ϕ is defined as the (second-order identity) predicate such that $\phi(\rho) = \rho, \forall$ first-order formula ρ. The instance of interest becomes:

$$(\forall \text{ formula } \delta \in \mathcal{E}, (\forall \text{ formula } \gamma \in \mathcal{E}, \gamma <_f \delta \Rightarrow \gamma) \Rightarrow \delta) \Rightarrow \forall \text{ formula } \rho \in \mathcal{E},\ \rho$$

The term-based Noetherian induction principle can be trivially represented as an instance of the formula-based Noetherian induction principle for which i) all elements from $\mathcal{E}$ are instances of the same formula ϕ, and ii) for any two instances $\phi(\overline{k})$ and $\phi(\overline{m})$ from $\mathcal{E}$, we have $\phi(\overline{k}) <_f \phi(\overline{m})$ if $\overline{k} <_t \overline{m}$.

Let us consider the 'P and Q' example, presented in (Wirth, 2004) and concerning the mutually dependent inductive definitions of the predicates P and Q:

$$\Rightarrow P(0) \quad (1) \qquad \Rightarrow Q(x,0) \quad (3)$$

$$P(x) \wedge Q(x,s(x)) \Rightarrow P(s(x)) \quad (2) \qquad P(x) \wedge Q(x,y) \Rightarrow Q(x,s(y)) \quad (4)$$

Both P and Q have naturals as arguments. Assuming the conjectures $P(u)$ and $Q(x,y)$, the direct application of the Peano principle on any of them, whatever is the induction variable, will fail to finish the proof. This is because the proof of $P(u)$ needs an instance of $Q(x,y)$, and vice versa. However, a successful proof can be built indirectly by firstly proving as additional lemma the conjunction $P(x) \wedge Q(x,y)$ of the two initial conjectures when the variable x is shared. It can be noticed that the proof of each initial conjecture can be trivially derived from the lemma. The difficulty of this approach is to build such lemmas as well as successful induction schemas for their proof. For example, the proof attempts for similar conjunctions, as $P(y) \wedge Q(x,y)$ (when the variable y is shared) and $P(u) \wedge Q(x,y)$ (when no variable is shared), will fail.

A more natural proof, based only on variable instantiations and unrolls of the definitions of P and Q, can result from using the lazy and mutual induction reasoning featured by the formula-based Noetherian induction methods to stop the proof development of some formulas, in particular for formulas that are instances of previously generated formulas. For example, the proof of $Q(x,y)$ starts by instantiating y by 0 and $s(y')$. $Q(x,0)$ is true by (3). The instance $Q(x,s(y'))$ is also true if $P(x)$ and $Q(x,y')$ are true, by (4). No transformation is supported by $Q(x,y')$, as an instance of the initial conjecture and used as IH. The variable x of $P(x)$ is further instantiated by 0 and $s(x')$. $P(0)$ is true by (1), while $P(s(x'))$ is true if both $Q(x',s(x'))$ and $P(x')$ are true, by (2). $P(x')$ is an instance of $P(x)$ while $Q(x',s(x'))$ is an instance of the initial conjecture. The proof is illustrated in Fig. 1 as a graph. The formula from any node p holds if the formulas from the nodes pointed out by the *downward* arrows starting from p also hold. The arrows are labelled by the corresponding instantiating substitutions when these formulas are instances of the formula from p. There are also *upward* (dashed) arrows that link a formula ϕ to another formula to which ϕ is an instance. They are labelled by the instantiating substitutions written in boldface style.

This proof is simpler, due to the lack of additional lemmas and the lazy employment of the IHs. The variable instantiation schema is that used by the Peano principle, i.e., a natural variable is instantiated by 0 and $s(x')$, where x' is a fresh natural variable. On the other hand, the induction ordering over formulas should be defined and has to ensure that the IHs are soundly used. In Sections 4 and 5, we will show different ways to apply the formula-based Noetherian induction principle in order to validate the use of IHs. However, the mechanical certification of the formula-based Noetherian induction reasoning is non-trivial. We will present in the next section the formal tools for formalising and certifying it using Coq.

3. Formalising formula-based Noetherian induction proofs

The Coq formalisation of formula-based Noetherian induction proofs is based on ideas presented in (Stratulat, 2010; Stratulat and Demange, 2011), initially developed for certifying implicit induction proofs. Mainly, we associate to each formula a *measure value*

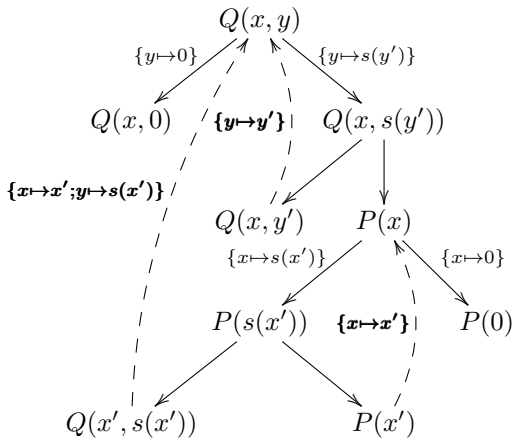

Fig. 1. The lazy and mutual induction proof of $Q(x,y)$.

that will help to compare formulas, hence to define the induction ordering $<_f$. Given a list LF of pairs of the form $(\phi,\ \mu_\phi)$ including the formulas to be proved and their corresponding measure values, the formula-based Noetherian induction principle can be reformulated as follows:

$$(\forall p \in \mathsf{LF}, (\forall p' \in \mathsf{LF}, snd(p') <_f snd(p) \Rightarrow fst(p')) \Rightarrow fst(p)) \Rightarrow \forall p \in \mathcal{E}, fst(p),$$

where the fst (resp., snd) function returns the first (resp., second) projection of a pair.

In Coq, the conditional part of the outermost implication can be formalised as:

```
Lemma main : ∀ F, In F LF → (∀ F', In F' LF → less (snd F') (snd F) → fst F') → fst F.
```

We assume that the induction ordering, denoted by less, is well-founded and stable under substitutions. In the following, less is assumed to implement $\prec\!\prec_{rpo}$. In this case, the measure value of a formula can be represented as the multiset of terms occurring in it. Given two pairs (ϕ_1,μ_{ϕ_1}) and (ϕ_2,μ_{ϕ_2}), we say that ϕ_1 is *smaller* than ϕ_2 if less $\mu_{\phi_1}\ \mu_{\phi_2}$ holds.

The main induction steps for proving the validity of the formula ϕ from each pair of LF are:

(1) the deduction part: choose the (instances of) formulas from LF as IHs that help proving ϕ;
(2) the ordering part: show that the chosen IHs in the deduction part are smaller than ϕ.

The ordering part can be omitted if the deduction part does not involve induction reasoning. The deduction part can be performed with different inference systems, as it will be shown in Sections 4, 5, and 6.

The less ordering has to be defined explicitly, for example, using the syntactic representations of terms provided by the COCCINELLE library (Contejean et al., 2007, 2010), a Coq library modelling mathematical notions for rewriting such as term algebras and induction orderings, or using the more general CoLoR library (Blanqui and Koprowski, 2011). Both libraries can be used independently or combined, but in the subsequent examples only COCCINELLE will be used.

3.1. Formalising the induction ordering and measure values with COCCINELLE

Formalising the induction ordering A COCCINELLE abstract term is recursively defined as:

```
Inductive term : Set :=
  | Var : variable → term
  | Term : symbol → list term → term.
```

COCCINELLE mutually defines $\prec_{rpo}$ and $\prec\!\!\prec_{rpo}$ (denoted by **rpo** and **rpo_mul**, respectively), together with other inductive predicates:

```
Inductive rpo (bb : nat) : term → term → Prop :=
| Subterm : ∀ f l t s, mem equiv s l → rpo_eq bb t s → rpo bb t (Term f l)
| Top_gt :
      ∀ f g l l', prec g f → (∀ s', mem equiv s' l' → rpo bb s' (Term f l)) →
        rpo bb (Term g l') (Term f l)
| Top_eq_lex :
      ∀ f g l l', status f = Lex → status g = Lex → prec_eq f g → (length l = length l'
∨ (length l' ≤ bb ∧ length l ≤ bb)) → rpo_lex bb l' l →
        (∀ s', mem equiv s' l' → rpo bb s' (Term g l)) →
        rpo bb (Term f l') (Term g l)
| Top_eq_mul :
      ∀ f g l l', status f = Mul → status g = Mul → prec_eq f g → rpo_mul bb l' l →
        rpo bb (Term f l') (Term g l)

with rpo_eq (bb : nat) : term → term → Prop :=
| Equiv : ∀ t t', equiv t t' → rpo_eq bb t t'
| Lt : ∀ s t, rpo bb s t → rpo_eq bb s t

with rpo_lex (bb : nat) : list term → list term → Prop :=
| List_gt : ∀ s t l l', rpo bb s t → rpo_lex bb (s :: l) (t :: l')
| List_eq : ∀ s s' l l', equiv s s' → rpo_lex bb l l' → rpo_lex bb (s :: l) (s' :: l')
| List_nil : ∀ s l, rpo_lex bb nil (s :: l)

with rpo_mul ( bb : nat) : list term → list term → Prop :=
| List_mul : ∀ a lg ls lc l l',
      permut0 equiv l' (ls ++ lc) → permut0 equiv l (a :: lg ++ lc) →
      (∀ b, mem equiv b ls → ∃ a', mem equiv a' (a :: lg) ∧ rpo bb b a') →
      rpo_mul bb l' l.
```

Both **rpo** and **rpo_mul** take a natural argument *bb* representing the maximal number of arguments of a function and used for proving its termination. **equiv** (resp., **permut0**) is the inductive predicate that checks if two terms are equivalent (resp. two lists of terms are permutable). length (resp., mem) is the usual function computing the length of a list (resp., whether a term is member of a list of terms).

The definitions of the *status*, *prec* and *prec_eq* functions are problem dependent. For the 'P and Q' example, they (and other intermediary functions) are defined as:

```
Inductive symb : Set :=
  | id_0
  | id_S
  | id_P
  | id_Q
```

```
Definition status (f:symb) :=
  match f with
    | id_0 ⇒ Mul
    | id_S ⇒ Mul
    | id_P ⇒ Mul
    | id_Q ⇒ Mul
  end.
```

```
Definition index (f:symb) :=
  match f with
    | id_0 ⇒ 2
    | id_S ⇒ 3
    | id_P ⇒ 9
    | id_Q ⇒ 9
  end.
```

```
Definition prec_bool (x y:A) : bool :=
  blt_nat (index x) (index y).

Definition prec (x y:A) :=
  prec_bool x y = true.

Definition prec_eq (x y:A) : Prop :=
  index x = index y.
```

The abstract COCCINELLE terms become concrete by defining the inductive set for function symbols (symbol), denoted above by **symb**. The precedence prec defined over **symb** is based on a (index) function that associates a natural value to each function symbol. Two symbols are equivalent if and only if they have the same indexes. The well-foundedness property of prec can now be formalised as:

```
Theorem prec_wf: well_founded prec.
```

The 'stability under substitutions' and well-foundedness properties of **rpo_mul** are scripted as:

```
Theorem rpo_mul_subst : ∀ A B: (list term), ∀ bb:nat, rpo_mul bb A B →
  ∀ σ, rpo_mul bb (map (apply_subst σ) A) (map (apply_subst σ) B).

Theorem wf_rpo_mul : well_founded prec → ∀ bb, well_founded (rpo_mul bb).
```

Finally, the less induction ordering is defined as an instance of **rpo_mul** initialising some internal data structures. For example, the value given as argument to **rpo** and **rpo_mul** is set to the constant max_size (usually a big natural value):

```
Notation less := (rpo_mul (bb (empty_rpo_infos max_size))).
```

Formalising the measure values Formulas and their measure values should share variables such that whenever a formula ϕ changes by instantiation, its measure value

μ_ϕ changes accordingly. The pair (ϕ, μ_ϕ) can be represented as the anonymous function `fun` $\overline{x} \Rightarrow (\phi, \mu_\phi)$, where μ_ϕ is the measure of ϕ formalised as a list of COCCINELLE terms and $\overline{x}$ is the vector of universally quantified variables shared between μ_ϕ and ϕ.

The process for converting terms from the LF formulas into COCCINELLE terms can be fully automatised, as follows:

- for each inductive set representing a sort employed in the specification, a new *model* function translating its constructor terms can be defined. For the 'P and Q' example, the translation function defined for the **nat** sort is:

```
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.
```

- the COCCINELLE counterpart of any function or predicate symbol f will be denoted by the symbol id_f prefixed by Term. The arguments of id_f are represented as a Coq list;
- the COCCINELLE counterpart of any variable x of sort s will be represented by the term (model_s x).

Example 3.1. The measure value used for (**Q** x 0) in the formula-based Noetherian induction proof from Fig. 1 can be represented as the following COCCINELLE term list: (model_nat x :: model_nat x :: (Term id_0 nil) :: nil), corresponding to the literate translation of the multiset $\{x, x, 0\}$.

3.2. Proving formulas from LF

The LF list from the main lemma should be adapted to include anonymous functions instead of pairs. The new LF should have the type of the form:

`Definition` type_LF := *argument_sort* → (`Prop` × (**List.list term**)),

where *argument_sort* is the sort written in a curried form of the most general version of the vector of shared variables allowing to define each anonymous function from LF.

By using the new LF definition, the main lemma becomes:

```
Lemma main : ∀ F, In F LF → ∀ u, (∀ F', In F' LF → ∀ u', less (snd (F' u')) (snd (F u))
→ fst (F' u')) → fst (F u).
```

All formulas from the LF list can be automatically certified by proving the all_true theorem, based on the main lemma and the general Noetherian induction principle built in Coq:

```
Theorem all_true: ∀ F, In F LF → ∀ u: nat, fst (F u).
```

As a case study, the methodology will be used in Sections 4 and 5 for certifying proofs of conjectures about conditional specifications.

3.3. Conditional specifications

Syntax We consider many-sorted conditional specifications consisting of a set of axioms Ax representing *equalities* of the form $\bigwedge_i l_i = r_i \Rightarrow l = r$, where '=' is the only predicate symbol and l and r, as well as l_i and r_i ($i \geq 0$), are two terms of same sort from $\mathcal{T}(\mathcal{F}, \mathcal{V})$. In addition, $(Var(l_i) \cup Var(r_i)) \subseteq (Var(l) \cup Var(r))$, for any i. The conjectures to be proved are also equalities. When $i = 0$, the equality is *unconditional* and the $\Rightarrow$ symbol is omitted, otherwise it is *conditional* and the equalities from the lhs of $\Rightarrow$ are called *conditions*. An equality $\bigwedge_i l_i = r_i \Rightarrow l = r$ is a *tautology* if $l = r$ is either of the form $t = t$, for some term t, or has a condition syntactically equal to $l = r$ or $r = l$. In the following, $\leq$ will denote a well-founded and 'stable under substitutions' quasi-ordering over instances of equalities.

Semantics We consider term-based (Herbrand) models and denote by M the unique *initial* (minimal) model of Ax (Baader and Nipkow, 1998). We say that an equality e is *true*, or it is an (initial) *consequence* of Ax and written as $Ax \models e$, if it is valid in M. A *counterexample* is any ground equality e such that $Ax \not\models e$. An equality is *false* or *has a counterexample* if there is a ground instance of it that is a counterexample. Extended to a set of equalities Φ, we say that Φ has a counterexample if there is an equality from Φ that has a counterexample.

4. Certifying implicit induction proofs

4.1. Implicit induction inference systems

An implicit induction inference system consists of *inference rules* representing transitions between pairs of sets of equalities of the form (E, H) called *states*, where E are *conjectures* and H are *premises*. By applying an inference rule, one of the conjectures, called *processed conjecture*, is firstly transformed into a (potentially empty) set of *new conjectures*, then it may be added to the set of premises in order to participate to further transformations. An *I-derivation* of a set of equalities E^0 is built from the list of states generated by a successive application of the rules of an inference system I, of the form $(E^0,\ \emptyset) \vdash_I (E^1,\ H^1) \vdash_I \ldots$. An *I-proof* of a set of equalities E^0 is a finite $(n+1)$-state derivation that ends in a state with an empty set of conjectures, of the form $(E^0,\ \emptyset) \vdash_I (E^1,\ H^1) \vdash_I \ldots \vdash_I (\emptyset,\ H^n)$. We assume that, during the construction of a proof derivation, $\leq$ is unique and defined over the instances of equalities from the proof derivation.

The implicit induction inference systems are *reductive*, i.e., at every derivation step, any new conjecture should be smaller, and sometimes smaller or equal, than some instance of the processed conjecture. Different sound reductive systems are presented in an *abstract* form because the way the new conjectures are created is ignored, for example, the inference system A (Stratulat, 2001). It has been shown that A generalizes many of the existing implicit induction procedures, for example the Implicit Induction procedure from (Bronsard et al., 1994), which is a generalization of the *hierarchical induction* procedure from (Reddy, 1990) and of the inductive procedures for conditional equalities from (Kounalis and Rusinowitch, 1990; Bronsard and Reddy, 1991; Bouhoula et al., 1995).

In the following, we will consider only instances of the inference system A', which is a simplified version of A.

ADDPREMISE: $(E \cup \{\phi\},\ H) \vdash_{A'} (E \cup \Phi,\ H \cup \{\phi\})$,
if, for any counterexample $\phi\tau$ of ϕ, there is a counterexample ψ in
i) $E \cup \Phi$ such that $\psi < \phi\tau$, or
ii) H such that $\psi \leq \phi\tau$.

SIMPLIFY: $(E \cup \{\phi\},\ H) \vdash_{A'} (E \cup \Phi,\ H)$,
if, for any counterexample $\phi\tau$ of ϕ, there is a counterexample ψ in $E \cup \Phi \cup H$ such that $\psi \leq \phi\tau$.

A' consists of two rules, ADDPREMISE and SIMPLIFY, that replace the processed conjecture ϕ by the set of new conjectures Φ. Moreover, ADDPREMISE adds ϕ to the set of premises if some ordering constraints are satisfied between counterexamples of ϕ and counterexamples in Φ and other equalities from the current state. On the other hand, SIMPLIFY does not add ϕ as a premise but allows for less restrictive ordering constraints.

Theorem 4.1 (soundness of A')**.** For any proof $(E^0,\ \emptyset) \vdash_{A'} \ldots \vdash_{A'} (\emptyset,\ H^n)$ using a set of axioms Ax, we have $Ax \models E^0$.

Proof. Let n be an arbitrary but fixed natural number, $(E^0,\ \emptyset) \vdash_{A'} \ldots \vdash_{A'} (\emptyset,\ H^n)$ a proof and assume by contradiction that E^0 has a false equality, hence a counterexample. By the well-foundedness property of the quasi-ordering $\leq$, there exists a minimal counterexample in the set of equalities generated during the proof. Since the proof ends with an empty set of conjectures, there is a last state $(E \cup \{\phi\},\ H)$ in the proof such that ϕ has a minimal counterexample $\phi\tau$. Some rule should be applied on ϕ, hence we perform a case analysis on the kind of this rule to show that no rule can be applied on ϕ, which contradicts the fact that the proof ends with an empty set of conjectures.

- **Case 1** Let us assume that the rule is ADDPREMISE. By considering the applicability conditions of ADDPREMISE for the particular case of $\phi\tau$, it can be noticed that the only possible solution is when H has a minimal counterexample ψ such that $\psi \sim \phi\tau$. Since there proof started with an empty set of premises, there should be a step in the proof when ADDPREMISE was applied on a state of the form $(E' \cup \{\phi'\},\ H')$, where H' has no minimal counterexamples equivalent to $\phi\tau$ but ϕ' has a minimal counterexample equivalent to $\phi\tau$. As previously, from the applicability conditions of ADDPREMISE, there should be a premise in H' having a minimal counterexample ψ' such that $\psi' \sim \phi\tau$, hence contradiction.
- **Case 2** Let us assume that the rule is SIMPLIFY. By analysing the applicability conditions of SIMPLIFY for $\phi\tau$ and the fact that $E \cup \Phi$ can not have minimal counterexamples equivalent to $\phi\tau$, we conclude that H should have a premise with a minimal counterexample ψ such that $\psi \sim \phi\tau$. This leads to a contradiction as in the previous case.

□

An abstract inference system becomes *concrete* by defining how the new conjectures from every proof step are built by the means of different reasoning techniques. Any concrete rule should satisfy the ordering constraints defined by some abstract rule.

We can define now a minimalistic and concrete implicit induction inference system, denoted by I^b, able to prove the conjecture from the 'P and Q' example, presented at the end of Section 2.

INSTNAT (I): $(E \cup \{\phi\langle x\rangle\},\ H) \vdash_{I^b} (E \cup \{\phi\{x \mapsto 0\}, \phi\{x \mapsto s(x')\}\},\ H)$,
where x' is a fresh variable.

DELINST (D): $(E \cup \{\phi\},\ H) \vdash_{I^b} (E,\ H)$,
if $\exists \psi \in H \cup Ax$ and a substitution σ such that $\phi \equiv \psi\sigma$.

REDEQ (R): $(E \cup \{\phi\},\ H) \vdash_{I^b} (E \cup \bigcup_i \{l_i\sigma = r_i\sigma\},\ H \cup \{\phi\})$,
if there is an axiom $\bigwedge_i l_i = r_i \Rightarrow l = r$ and a substitution σ such that $\phi \equiv (l\sigma = r\sigma)$.

INSTNAT replaces an equality ϕ with a natural variable x by two equalities derived from ϕ by instantiating ϕ with 0 and the successor of a fresh natural variable, respectively. DELINST deletes the processed conjecture if it is an instance of a premise or axiom. Finally, REDEQ replaces any unconditional equality that matches the conclusion of a conditional axiom with the set of the corresponding instances of the conditions of the conditional axiom. In addition, the processed conjecture is added as premise.

The equational axioms for the 'P and Q example' are built from the translation of the inductive definitions of the predicates P and Q, by considering them as the boolean functions p and q, respectively, and defined as:

$$p(0) = true \tag{5}$$
$$p(x) = true \wedge q(x, s(x)) = true \Rightarrow p(s(x)) = true \tag{6}$$
$$p(x) = false \Rightarrow p(s(x)) = false \tag{7}$$
$$q(x, s(x)) = false \Rightarrow p(s(x)) = false \tag{8}$$

$$q(x, 0) = true \tag{9}$$
$$p(x) = true \wedge q(x, y) = true \Rightarrow q(x, s(y)) = true \tag{10}$$
$$p(x) = false \Rightarrow q(x, s(y)) = false \tag{11}$$
$$q(x, y) = false \Rightarrow q(x, s(y)) = false \tag{12}$$

where $false$ and $true$ are the usual boolean constants.

In order to define the ordering $<$ for this example, the measure value of an equality of the form $\bigwedge_i l_i = r_i \Rightarrow l = r$ is defined as the multiset of terms $\bigcup_i |l_i| \cup \bigcup_i |r_i| \cup |l| \cup |r|$, where $|t|$ is defined as

- $\{x, x\}$ if t is of the form $p(x)$,
- $\{x, x, y\}$ if t is of the form $q(x, y)$,
- $\{t\}$, otherwise.

The ordering used for comparing multisets of terms is the multiset extension of the rpo using the precedence $false <_{\mathcal{F}} true <_{\mathcal{F}} 0 <_{\mathcal{F}} s$. It can be noticed that for any conditional axiom defining p and q, of the form $\bigwedge_i e_i \Rightarrow e$, we have that $e_i < e$. This property will be useful for the sound application of the RedEq rule.

The I^b-proof of the equality $q(x, y) = true$ is:

$$
\begin{array}{ll}
(\{\underline{q(x,y) = true}\}, \emptyset) & \vdash^{I}_{I^b} \\
(\{\underline{q(x,0) = true}, q(x, s(y')) = true\}, \emptyset) & \vdash^{D}_{I^b} \\
(\{\underline{q(x, s(y')) = true}\}, \emptyset) & \vdash^{R}_{I^b} \\
(\{\underline{p(x) = true}, q(x, y') = true\}, \{q(x, s(y')) = true\}) & \vdash^{I}_{I^b} \\
(\{\underline{p(0) = true}, p(s(x')) = true, q(x, y') = true\}, \{q(x, s(y')) = true\}) & \vdash^{D}_{I^b} \\
(\{\underline{p(s(x')) = true}, q(x, y') = true\}, \{q(x, s(y')) = true\}) & \vdash^{R}_{I^b} \\
(\{p(x') = true, \underline{q(x', s(x')) = true}, q(x, y') = true\}, & \\
\qquad \{p(s(x')) = true, q(x, s(y')) = true\}) & \vdash^{D}_{I^b} \\
(\{\underline{p(x') = true}, q(x, y') = true\}, \{p(s(x')) = true, q(x, s(y')) = true\}) & \vdash^{I}_{I^b} \\
(\{\underline{p(0) = true}, p(s(x'')) = true, q(x, y') = true\}, & \\
\qquad \{p(s(x')) = true, q(x, s(y')) = true\}) & \vdash^{D}_{I^b} \\
(\{\underline{p(s(x'')) = true}, q(x, y') = true\}, \{p(s(x')) = true, q(x, s(y')) = true\}) & \vdash^{D}_{I^b} \\
(\{\underline{q(x, y') = true}\}, \{p(s(x')) = true, q(x, s(y')) = true\}) & \vdash^{I}_{I^b} \\
(\{\underline{q(x,0) = true}, q(x, s(y'')) = true\}, \{p(s(x')) = true, q(x, s(y')) = true\}) & \vdash^{D}_{I^b} \\
(\{\underline{q(x, s(y'')) = true}\}, \{p(s(x')) = true, q(x, s(y')) = true\}) & \vdash^{D}_{I^b} \\
(\emptyset, \{p(s(x')) = true, q(x, s(y')) = true\}) &
\end{array}
$$

where the underlined equalities are the processed conjectures from each proof state.

Theorem 4.2 (soundness of I^b)**.** The inference system I^b is sound.

Proof. It follows directly from Theorem 4.1 if we prove that each I^b-rule is the instance of an A'-rule.

- InstNat is an instance of Simplify because if the processed conjecture $\phi\langle x\rangle$ has a counterexample then it should be in the set of new conjectures $\{\phi\{x \mapsto 0\}, \phi\{x \mapsto s(x')\}\}$
- DelInst is an instance of Simplify because whenever the equality instantiated by the processed conjecture ϕ is
  · an axiom, then ϕ has no counterexamples, or
  · a premise ψ from the current state, then any counterexample $\phi\tau$ is an instance of ψ.
- RedEq, using the conditional axiom $\bigwedge_i l_i = r_i \Rightarrow l = r$ for which there is a substitution σ such that the processed conjecture ϕ is $(l\sigma = r\sigma)$, is an instance of AddPremise because whenever ϕ has a counterexample $\phi\tau$, one of the equalities from the multiset $\bigcup_i \{l_i\sigma\tau = r_i\sigma\tau\}$ is a counterexample smaller than $\phi\tau(\equiv (l\sigma\tau = r\sigma\tau))$, by the fact that $l_i = r_i < l = r$, for each i, and the 'stability under substitutions' property of $<$.

□

Thanks to Theorem 4.2, we can conclude that $q(x, y) = true$ is a consequence of the axioms defining p and q.

4.2. A first attempt to certify the proof of $q(x, y) = true$

A translation of the functions p and q in Coq, using a functional programming style, can be

```
Fixpoint p (u:nat) : bool :=
match u with
  | 0 ⇒ true
  | (S u') ⇒ if andb (p u') (q u' (S u')) then true else false
end
with
q (x y: nat): bool :=
  match y with
    | 0 ⇒ true
    | (S y') ⇒ if andb (p x) (q x y') then true else false
  end.
```

Unfortunately, the version of Coq used for certification, labelled as 8.4pl4, is not able to prove automatically the termination of the P and Q functions, yielding the message `Error: Cannot guess decreasing argument of fix`. Moreover, specifying user-defined well-founded orderings for proving that an argument is decreasing is not allowed for mutually recursive functions, as it is clearly stated by the message `Error: Cannot use mutual definition with well-founded recursion or measure`. We will show in Subsection 5.3 how to handle this situation using the logic programming style.

4.3. Certifying implicit induction proofs concerning specifications that are convertible into valid Coq script

In the rest of the section, we will stick to certifying implicit induction proofs that involve equational specifications convertible into valid Coq script, by using a functional programming style. Let us consider the function symbols *even* and *odd*, recursively defined over naturals, as:

$$even(0) = true \quad (13) \qquad odd(0) = false \quad (15)$$
$$even(s(x)) = odd(x) \quad (14) \qquad odd(s(x)) = even(x) \quad (16)$$

as well as the addition over naturals, denoted by '+' and defined by the axioms:

$$0 + x = x \quad (17) \qquad s(x) + y = s(x + y) \quad (18)$$

The Coq translation of the equational definitions yields functions whose termination can be automatically checked:

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

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

We can go further and try to prove the more complex conjecture

$$odd(u_1 + u_2) = true \wedge even(u_2 + u_3) = true \Rightarrow odd(u_1 + u_3) = true \tag{19}$$

by using reductive reasoning techniques. *Rewriting* is a most effective reductive technique for reasoning on equational specifications. When dealing with equality reasoning, the reductive constraints between (instances of) the processed conjecture and new conjectures can be *implicitly* satisfied if the equational specifications are represented in terms of rewrite systems and the IHs are orientable. In some cases, the constraints related to IHs can be partially weakened. For example, (Stratulat, 2008) proposes a method allowing for *relaxed* rewriting (Bouhoula et al., 1995) to deal with unorientable IHs by integrating explicit induction schemas. It covers the *term* (Reddy, 1990), *ordered* (Dershowitz and Reddy, 1993), *enhanced* and *incremental* (Aoto, 2006) rewriting induction procedures.

Given a reduction ordering $\prec$ over terms, any equality $l = r$ can be oriented into the *rewrite rule* $l \rightarrow r$ if $r \prec l$. Let us assume a set ρ of rewrite rules. A term u can be rewritten to u' by the *rewrite operation* $u \rightarrow_\rho u'$ if there are a rewrite rule $l = r \in \rho$ and a substitution σ such that $l\sigma$ is a subterm of u. The rewrite operation builds u' from u by replacing the subterm $l\sigma$ by $r\sigma$. By abuse of notation, $\rightarrow_\rho$ is extended to rewrite conditional equalities: if a conditional equality e' of the form $\bigwedge_i l_i = r_i \Rightarrow l = r$ has a term $s \in \bigcup_i \{l_i, r_i\} \cup \{l, r\}$ that is rewritten to s' using rewrite rules from ρ then we write $e \rightarrow_\rho e'$, where e' derives from e by replacing s with s'.

Example 4.3. The axioms (13)-(18) can be oriented from left to right using as reduction ordering the rpo built from the precedence over the function symbols stating that $false <_\mathcal{F} true <_\mathcal{F} 0 <_\mathcal{F} s <_\mathcal{F} + <_\mathcal{F} even$ and $even \sim_\mathcal{F} odd$.

Let $\leq$ be the well-founded and 'stable under substitutions' quasi-ordering over the equalities whose strict part is the multiset extension $\prec\!\!\prec$ of the reduction ordering $\prec$. The measure value of an equality of the form $\bigwedge_i l_i = r_i \Rightarrow l = r$ is defined for this example as the multiset of terms $\bigcup_i \{l_i, r_i\} \cup \{l, r\}$.

Theorem 4.4 (reductiveness of rewriting)**.** Let ρ be a set of rewrite rules and e, e' two equalities. If $e \rightarrow_\rho e'$, then $e' \prec\!\!\prec e$.

Proof. Let us assume that e is of the form $\bigwedge_i l_i = r_i \Rightarrow l = r$ and that $s \in \bigcup_i \{l_i, r_i\} \cup \{l, r\}$ was rewritten to s' by a rewrite rule $g \rightarrow d$ from ρ. Then, there is a substitution σ

such that $g\sigma$ is a subterm of s. By the 'stability under substitutions' property of $\prec$, we have that $d\sigma \prec g\sigma$, and by the 'stability under contexts' property, $s' \prec s$ holds.

On the other hand, the measure value of $\bigwedge_i l_i = r_i \Rightarrow l = r$ is the multiset $\bigcup_i \{l_i, r_i\} \cup \{l, r\}$, s being one of its elements. By the definition of the multiset extension relation, the replacement of s by s' in this multiset yields a smaller multiset, hence $e' \prec\prec e$. □

Proofs of the conjecture (19) can be built using the inference system I^f:

GenNat (G): $(E \cup \{\phi\langle x\rangle\},\ H) \vdash_{I^f} (E \cup \{\phi_1, \phi_2\},\ H \cup \{\phi\})$,
where $\phi\{x \mapsto 0\} \rightarrow_{Ax} \phi_1$, $\phi\{x \mapsto s(x')\} \rightarrow_{Ax} \phi_2$ and x' is a fresh variable.

SimpEq (S): $(E \cup \{\phi\},\ H) \vdash_{I^f} (E \cup \Phi,\ H)$,
if either i) ϕ is a tautology; in this case Φ is empty;
or, ii) $\phi \rightarrow_{Ax \cup (E \cup \Phi \cup H)_{\leq \phi}} \psi$; in this case, Φ is $\{\psi\}$.

Subsumption (E): $(E \cup \{\phi\}, H) \vdash_{I^f} (E,\ H)$,
if ϕ is an instance of an equality from H.

Negative Clash (N): $(E \cup \{\phi\}, H) \vdash_{I^f} (E,\ H)$,
if ϕ is a conditional equality and $true = false$ or $false = true$ is a condition of ϕ.

GenNat firstly instantiates a natural variable of the processed conjecture by 0 and the successor of a fresh natural variable, then rewrites the two instances with axioms, the results of the rewriting operations being stored as new conjectures. At the end, the processed conjecture is saved as a new premise. SimpEq either deletes the tautologies or performs rewrite operations on the processed conjecture with axioms or instances of equalities from the current state. Subsumption deletes the processed conjecture if it is an instance of a premise. Finally, Negative Clash deletes the conditional equalities having a false condition of the form $true = false$ or $false = true$.

The I^f-proof of (19) is more complex than that of the conjecture $q(x, y) = true$. For lack of space, the conditional equalities from this proof will be presented in a more compact way, as atoms. The list of atoms and their corresponding conditional equalities are:

$e\_13(u_1, u_2, u_3) : odd(u_1 + u_2) = true \wedge even(u_2 + u_3) = true \Rightarrow odd(u_1 + u_3) = true$
$e\_23(u_2, u_3) : odd(0 + u_2) = true \wedge even(u_2 + u_3) = true \Rightarrow odd(u_3) = true$
$e\_29(u_4, u_2, u_3) : odd(s(u_4) + u_2) = true \wedge even(u_2 + u_3) = true \Rightarrow$
$odd(s(u_4 + u_3)) = true$
$e\_36(u_2, u_3) : odd(u_2) = true \wedge even(u_2 + u_3) = true \Rightarrow odd(u_3) = true$
$e\_49(u_4, u_2, u_3) : even(u_4 + u_2) = true \wedge even(u_2 + u_3) = true \Rightarrow even(u_4 + u_3) = true$
$e\_67(u_3) : odd(0) = true \wedge even(u_3) = true \Rightarrow odd(u_3) = true$
$e\_73(u_4, u_3) : odd(s(u_4)) = true \wedge even(s(u_4 + u_3)) = true \Rightarrow odd(u_3) = true$
$e\_81(u_3) : false = true \wedge even(u_3) = true \Rightarrow odd(u_3) = true$
$e\_91(u_4, u_3) : even(u_4) = true \wedge odd(u_4 + u_3) = true \Rightarrow odd(u_3) = true$
$e\_113(u_2, u_3) : even(0 + u_2) = true \wedge even(u_2 + u_3) = true \Rightarrow even(u_3) = true$

$e\_119(u_5, u_2, u_3) : even(s(u_5) + u_2) = true \wedge even(u_2 + u_3) = true \Rightarrow$
$even(s(u_5 + u_3)) = true$
$e\_138(u_2, u_3) : even(u_2) = true \wedge even(u_2 + u_3) = true \Rightarrow even(u_3) = true$
$e\_189(u_3) : even(0) = true \wedge odd(u_3) = true \Rightarrow odd(u_3) = true$
$e\_195(u_5, u_3) : even(s(u_5)) = true \wedge odd(s(u_5 + u_3)) = true \Rightarrow odd(u_3) = true$
$e\_237(u_3) : even(0) = true \wedge even(u_3) = true \Rightarrow even(u_3) = true$
$e\_243(u_5, u_3) : even(s(u_5)) = true \wedge even(s(u_5 + u_3)) = true \Rightarrow even(u_3) = true$
$e\_267(u_5, u_3) : odd(u_5) = true \wedge odd(u_5 + u_3) = true \Rightarrow even(u_3) = true$
$e\_275(u_3) : false = true \wedge odd(0 + u_3) = true \Rightarrow even(u_3) = true$
$e\_295(u_3) : odd(0) = true \wedge odd(u_3) = true \Rightarrow even(u_3) = true$
$e\_301(u_6, u_3) : odd(s(u_6)) = true \wedge odd(s(u_6 + u_3)) = true \Rightarrow even(u_3) = true$
$e\_317(u_3) : false = true \wedge odd(u_3) = true \Rightarrow even(u_3) = true$

Using the notation with atoms, the proof can be represented as:

$(\{\underline{e\_13(u_1, u_2, u_3)}\}, \emptyset) \vdash^{G}_{I^c_f}$
$(\{\underline{e\_23(u_2, u_3), e\_29(u_4, u_2, u_3)}\}, \{e\_13(u_1, u_2, u_3)\}) \vdash^{*S}_{I^c_f}$
$(\{\underline{e\_36(u_2, u_3)}, e\_49(u_4, u_2, u_3)\}, \{e\_13(u_1, u_2, u_3)\}) \vdash^{G}_{I^c_f}$
$(\{\underline{e\_67(u_3), e\_73(u_4, u_3)}, e\_49(u_4, u_2, u_3)\}, \{e\_36(u_2, u_3), e\_13(u_1, u_2, u_3)\}) \vdash^{*S}_{I^f}$
$(\{\underline{e\_81(u_3)}, e\_91(u_4, u_3), e\_49(u_4, u_2, u_3)\}, \{e\_36(u_2, u_3), e\_13(u_1, u_2, u_3)\}) \vdash^{N}_{I^f}$
$(\{\underline{e\_73(u_4, u_3)}, e\_49(u_4, u_2, u_3)\}, \{e\_36(u_2, u_3), e\_13(u_1, u_2, u_3)\}) \vdash^{S}_{I^f}$
$(\{e\_91(u_4, u_3), \underline{e\_49(u_4, u_2, u_3)}\}, \{e\_36(u_2, u_3), e\_13(u_1, u_2, u_3)\}) \vdash^{G}_{I^f}$
$(\{e\_91(u_4, u_3), \underline{e\_113(u_2, u_3), e\_119(u_5, u_2, u_3)}\}, \{e\_49(u_4, u_2, u_3), e\_36(u_2, u_3), e\_13(u_1, u_2, u_3)\}) \vdash^{*S}_{I^f}$
$(\{e\_91(u_4, u_3), e\_138(u_2, u_3), \underline{e\_13(u_5, u_2, u_3)}\}, \{e\_49(u_4, u_2, u_3), e\_36(u_2, u_3), e\_13(u_1, u_2, u_3)\}) \vdash^{E}_{I^f}$
$(\{\underline{e\_91(u_4, u_3)}, e\_138(u_2, u_3)\}, \{e\_49(u_4, u_2, u_3), e\_36(u_2, u_3), e\_13(u_1, u_2, u_3)\}) \vdash^{G}_{I^f}$
$(\{\underline{e\_189(u_3)}, e\_195(u_5, u_3), e\_138(u_2, u_3)\},$
$\{e\_91(u_4, u_3), e\_49(u_4, u_2, u_3), e\_36(u_2, u_3), e\_13(u_1, u_2, u_3)\}) \vdash^{S}_{I^f}$
$(\{\underline{e\_195(u_5, u_3)}, e\_138(u_2, u_3)\}, \{e\_91(u_4, u_3), e\_49(u_4, u_2, u_3), e\_36(u_2, u_3), e\_13(u_1, u_2, u_3)\}) \vdash^{*S}_{I^f}$
$(\{\underline{e\_36(u_5, u_3)}, e\_138(u_2, u_3)\}, \{e\_91(u_4, u_3), e\_49(u_4, u_2, u_3), e\_36(u_2, u_3), e\_13(u_1, u_2, u_3)\}) \vdash^{E}_{I^f}$
$(\{\underline{e\_138(u_2, u_3)}\}, \{e\_91(u_4, u_3), e\_49(u_4, u_2, u_3), e\_36(u_2, u_3), e\_13(u_1, u_2, u_3)\}) \vdash^{G}_{I^f}$
$(\{\underline{e\_237(u_3)}, e\_243(u_5, u_3)\},$
$\{e\_138(u_2, u_3), e\_91(u_4, u_3), e\_49(u_4, u_2, u_3), e\_36(u_2, u_3), e\_13(u_1, u_2, u_3)\}) \vdash^{S}_{I^f}$
$(\{\underline{e\_243(u_5, u_3)}\}, \{e\_138(u_2, u_3), e\_91(u_4, u_3), e\_49(u_4, u_2, u_3), e\_36(u_2, u_3), e\_13(u_1, u_2, u_3)\}) \vdash^{*S}_{I^f}$
$(\{\underline{e\_267(u_5, u_3)}\}, \{e\_138(u_2, u_3), e\_91(u_4, u_3), e\_49(u_4, u_2, u_3), e\_36(u_2, u_3), e\_13(u_1, u_2, u_3)\}) \vdash^{G}_{I^f}$
$(\{\underline{e\_295(u_3), e\_301(u_6, u_3)}\},$
$\{e\_267(u_5, u_3), e\_138(u_2, u_3), e\_91(u_4, u_3), e\_49(u_4, u_2, u_3), e\_36(u_2, u_3), e\_13(u_1, u_2, u_3)\}) \vdash^{*S}_{I^f}$
$(\{\underline{e\_317(u_3)}, e\_138(u_6, u_3)\},$
$\{e\_267(u_5, u_3), e\_138(u_2, u_3), e\_91(u_4, u_3), e\_49(u_4, u_2, u_3), e\_36(u_2, u_3), e\_13(u_1, u_2, u_3)\}) \vdash^{N}_{I^f}$
$(\{\underline{e\_138(u_6, u_3)}\},$
$\{e\_267(u_5, u_3), e\_138(u_2, u_3), e\_91(u_4, u_3), e\_49(u_4, u_2, u_3), e\_36(u_2, u_3), e\_13(u_1, u_2, u_3)\}) \vdash^{E}_{I^f}$
$(\{\emptyset, \{e\_267(u_5, u_3), e\_138(u_2, u_3), e\_91(u_4, u_3), e\_49(u_4, u_2, u_3), e\_36(u_2, u_3), e\_13(u_1, u_2, u_3)\})$

As for the I^b-proof of the equality $q(x, y) = true$, the processed conjectures from each state are underlined. $\vdash^{*S}_{I^f}$ means that SimpEq was applied several times. The induction

reasoning occurs while executing the SUBSUMPTION rule, the instances of premises used as IHs being built by need.

Theorem 4.5 (soundness of I^f)**.** The inference system I^f is sound.

Proof. As for the proof of Theorem 4.2, we show that each I^f-rule is the instance of an A'-rule.

- GENNAT is an instance of ADDPREMISE. If the processed conjecture $\phi\langle x\rangle$ has a counterexample $\phi\tau$ then it should be in the set of intermediary conjectures $\{\phi\{x \mapsto 0\}, \phi\{x \mapsto s(x')\}\}$. Since the two intermediary conjectures are rewritten with axioms, the set of new conjectures has a counterexample smaller than $\phi\tau$, by the reductiveness property of rewriting and the 'stability under substitutions' property of $\leq$.
- SIMPEQ is an instance of SIMPLIFY. If the processed conjecture ϕ has a counterexample $\phi\tau$, then ϕ should be rewritten to ψ with i) axioms, or ii) other equalities from the current state or from the new set of conjectures which are smaller or equal than ϕ. We have that $\psi < \phi$ by the reductiveness property of rewriting and $\psi\tau < \phi\tau$ by the 'stability under substitutions' property of $<$. If $\psi\tau$ is false, then $\psi\tau$ is a counterexample from the new set of conjectures which is smaller than $\phi\tau$. If $\psi\tau$ is true, it means that the rewrite rule e was selected from the current state and satisfies $e \leq \phi$. By the 'stability under substitutions' property of $\leq$, we have that $e\tau \leq \phi\tau$. Moreover, $e\tau$ is a counterexample since $\phi\tau$ is false but $\psi\tau$ is true.
- SUBSUMPTION is an instance of SIMPLIFY because whenever the processed conjecture ϕ is an instance of a premise, any counterexample of ϕ is also a counterexample of that premise.
- NEGATIVE CLASH is an instance of SIMPLIFY because the processed conjecture has no counterexamples.

□

By Theorem 4.5, we conclude that the conjecture (19) is a consequence of the axioms defining *even*, *odd* and '+'.

4.4. Coq formalization and certification of implicit induction proofs based on functional programming style

The induction ordering is built similarly as for the 'P and Q' example, shown in Subsection 3.1, excepting that the precedence over the function symbols changes, as follows:

```
Inductive symb : Set :=
  | id_0
  | id_S
  | id_true
  | id_false
  | id_even
  | id_odd
  | id_plus.
```

```
Definition index (f : symb) :=
  match f with
  | id_0 ⇒ 2
  | id_S ⇒ 3
  | id_true ⇒ 4
  | id_false ⇒ 5
  | id_even ⇒ 10
  | id_odd ⇒ 10
  | id_plus ⇒ 7
  end.
```

The formula-based Noetherian induction principle will be applied on the set of all equalities encountered in the implicit induction proof of $e\_13(u_1, u_2, u_3)$. The LF list type_LF_13 and its type LF_13 are:

```
Definition type_LF_13 := nat → nat → nat → Prop × List.list term.
Definition LF_13 := [F_13, ... ,F_317], (* all equalities from the proof *)
```

where the anonymous functions from LF_13 are:

```
Definition F_13 : type_LF_13 :=
  fun u1 u2 u3 ⇒
  ( e_13(u1,u2,u3),
  Term id_odd (Term
                  id_plus (model_nat u1 :: model_nat u2 :: nil) :: nil)
  :: Term id_true nil
     :: Term id_even
          (Term id_plus
             (model_nat u2 :: model_nat u3 :: nil) :: nil)
        :: Term id_true nil
           :: Term id_odd
                (Term id_plus (
                                 model_nat u1 :: model_nat u3 :: nil) :: nil)
             :: Term id_true nil :: nil).

⋮

Definition F_317 : type_LF_13 :=
  fun u3 _ _ ⇒
  ( e_317(u3),
  Term id_false nil
  :: Term id_true nil
     :: Term id_odd (model_nat u3 :: nil)
        :: Term id_true nil
           :: Term id_even (model_nat u3 :: nil) :: Term id_true nil :: nil).
```

The measure value attached to any equality e in the corresponding anonymous function is the list of the literate representation in COCCINELLE of the terms from e.

The main lemma becomes:

```
Lemma main_13 : ∀ F, In F LF_13 → ∀ u1 u2 u3, (∀ F', In F' LF_13 → ∀ e1 e2 e3,
  less (snd (F' e1 e2 e3)) (snd (F u1 u2 u3)) → fst (F' e1 e2 e3)) → fst (F u1 u2
u3).
```

The proof of the main_13 lemma starts with a case analysis on the anonymous functions from LF_13. We describe the scenarios for building the Coq script by translating every single implicit induction inference step. Given an anonymous function $F \in$ LF_13,

- the main steps for generating the Coq script for the case when a GenNat rule is applied on an equality e from F are:

(1) *generation and application of the instantiation schema.* The instantiation schema of variables from e consists in replacing a variable v of natural sort with 0 and successor of a fresh variable. In Coq, this can be easily performed with the `destruct` tactic applied on v. However, an instantiation schema reproduced by the means of *functional schemes* (Barthe and Courtieu, 2002) is more flexible than the instantiation schemas issued from the definitions of inductive sets on which the `destruct` tactic are based:

```
Fixpoint f (u1: nat) {struct u1} : nat :=
 match u1 with
   | 0 ⇒ 0
   | (S u2) ⇒ 0
end.
```

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

The functional scheme can be applied on any natural variable u_1, as follows:

`pattern` u_1, (`f` u_1). `apply f_ind.`

(2) *validation of each instance and ordering constraint.* For each equality instance ϕ generated during the application of GenNat, we assign to F' the anonymous function from LF_13 that corresponds to the equality resulted from the rewriting of ϕ in the I^f-proof. If the axioms are put in the Coq rewrite base, the logical equivalence between ϕ and the rewritten instance can be automatically checked by the `auto` tactic. The ordering constraints requiring that the measure value of the equality from F' be smaller than μ_ϕ are also automatically checked by user-defined tactics. The *rewrite_model* tactic simplifies μ_ϕ by unfolding the model_nat translation functions on the subterms of ϕ of the form (model_nat (S u)). The *solve_rpo_mul* tactic i) replaces the terms of the form (model_nat u) by COCCINELLE variables, ii) performs the comparison test, representing a test case for checking Theorem 4.4, and iii) by the 'stability under substitutions' property of **rpo_mul**, preserves the comparison result for the instance built with the substitution mapping the COCCINELLE variables to the corresponding (model_nat u) terms.

- the scenario corresponding to the application of a SimpEq rule using rewriting is similar to that presented at the step (2) of the scenario built for GenNat. The tautologies are eliminated by the `auto` tactic;
- the application of a Negative Clash corresponds to the application of the `discriminate` tactic;
- the Subsumption steps are ignored because the equality from the anonymous function F is an instance of an equality from another anonymous function from LF_13.

Next, we show that any formula from LF_13 is true:

```
Theorem all_true_13 :  ∀ F,  In F LF_13' →
  ∀ (u1 : nat) (u2 : nat) (u3 : nat), fst (F u1 u2 u3).
```

Finally, our property is certified:

```
Theorem true_13 :  ∀ (u1 : nat) (u2 : nat) (u3 : nat),  odd (plus u1 u2) = true →
  even (plus u2 u3) = true → odd (plus u1 u3) = true.
```

5. Certifying cyclic proofs

In the previous section, we have shown that the computational cost for certifying implicit induction proofs also depends on the number of equalities encountered in the proofs. In practice, it is common that implicit induction provers (automatically) generate large proofs, with hundreds or thousands of equalities (Barthe and Stratulat, 2003; Rusinowitch et al., 2003). In this case, the proof certification effort becomes non-negligeable. We present a different method for proving conjectures about conditional specifications, based on a reductive-free cyclic induction approach proposed in (Stratulat, 2012) for which the certification process is more effective. The proofs are built by outlining the non-trivial induction reasoning in terms of *cycles* of equalities. Compared to the validation process of implicit induction proofs from Section 4, the validation of cyclic proofs needs fewer ordering constraints and shorter LF lists.

5.1. Cyclic induction inference systems

We introduce the abstract inference system D', similar to the system D (Stratulat, 2012), which consists of the following three rules:

DEDUCTION: $E \cup \{\phi\} \vdash_{D'} E \cup \Phi$,
if Φ has a counterexample whenever ϕ has a counterexample.

SPLIT: $E \cup \{\phi\} \vdash_{D'} E \cup \Phi$,
if Φ is a set of instances of ϕ such that Φ has a counterexample whenever ϕ has a counterexample.

INDUCTION: $E \cup \{\phi\} \vdash_{D'} E \cup \Phi$,
if there exists an equality ψ previously generated in the derivation such that Φ or ψ have a counterexample whenever ϕ has a counterexample.

Compared to the implicit induction inference rules, the D'-rules are transitions between multisets of equalities that transform an equality (i.e., the processed conjecture) into a set of new equalities (i.e., new conjectures). DEDUCTION ensures that for any counterexample of the processed conjecture there is a counterexample in the set of new conjectures. SPLIT is a particular case of DEDUCTION, requiring that the set of new conjectures consists of instances of the processed conjecture. Finally, INDUCTION is the only rule that performs induction reasoning, by allowing instances of previously generated equalities in the derivation to be used as IHs when transforming the processed conjecture. It can be seen as a generalisation of DEDUCTION since for any counterexample of the processed conjecture we may not require for a counterexample in the set of new conjectures if the equality used as IH has already a counterexample.

Definition 5.1 $(D' - preproof)$**.** Given a multiset of equalities E^0, any finite derivation of the form $E^0 \vdash_{D'} \ldots \vdash_{D'} \emptyset$ is a D'-preproof of E^0.

Concrete inference rules can be built by showing how the new conjectures from the D'-rules are generated using specific reasoning techniques. Based on the reasoning techniques employed by the inference system I^b from Section 4, we can build the inference system I_c^b:

DelInst' (D_c): $E \cup \{\phi\} \vdash_{I_c^b} E$,
if there are $\psi \in Ax$ and a substitution σ such that $\phi \equiv \psi\sigma$.

RedEq' (R_c): $E \cup \{\phi\} \vdash_{I_c^b} E \cup \bigcup_i \{l_i\sigma = r_i\sigma\}$,
if $\bigwedge_i l_i = r_i \Rightarrow l = r \in Ax$ and $\exists\sigma$ such that $\phi \equiv (l\sigma = r\sigma)$ or $\phi \equiv (r\sigma = l\sigma)$.

SplitNat (S_c): $E \cup \{\phi\langle x\rangle\} \vdash_{I_c^b} E \cup \{\phi\{x \mapsto 0\}, \phi\{x \mapsto s(x')\}\}$,
where x' is a fresh variable.

IndNat (I_c): $E \cup \{\phi\} \vdash_{I_c^b} E$,
if there are a previously generated equality ψ and a substitution σ such that $\phi \equiv \psi\sigma$.

DelInst' deletes the processed conjecture if it is an instance of an axiom. RedEq' firstly checks whether the processed conjecture is an instance of the conclusion of some axiom, then adds as new conjectures the set of corresponding instances of the conditions of the axiom. SplitNat applies on conjectures with natural variables and replaces them by their instances resulted by replacing some natural variable with 0 and successor of a new variable. Finally, IndNat deletes the processed conjecture if it is an instance of a previous conjecture in the derivation.

Theorem 5.2. Any I_c^b-rule is an instance of a D'-rule.

Proof. We perform a case analysis on the I_c^b-rules:
- DelInst' is an instance of Deduction for the case when the set of new conjectures is empty since the processed conjecture has no counterexamples;
- RedEq' is an instance of Deduction because for any counterexample of the processed conjecture ϕ there is one in the set of corresponding instances of the equality conditions of the axiom whose conclusion was instantiated by ϕ;
- SplitNat is an instance of Split since any counterexample of the processed conjecture is also a counterexample in the set of new conjectures;
- IndNat is an instance of Induction because any counterexample of the processed conjecture is also a counterexample of the previous equality whose instance was used as IH.

□

An I_c^b-preproof of a multiset of equalities E^0 is any finite I_c^b-derivation that starts with E^0 and finishes with an empty set of equalities.

Example 5.3. The following I_c^b-preproof of $\{q(x, y) = true\}$ can be built by using the axioms (5)-(12) and the proof scenario given at the end of Section 2:

$\{\underline{q(x,y) = true}\} \vdash^{S_c}_{I^b_c} \{\underline{q(x,0) = true}, q(x, s(y')) = true\} \vdash^{D_c}_{I^b_c} \{\underline{q(x, s(y')) = true}\} \vdash^{R_c}_{I^b_c}$
$\{\underline{q(x,y') = true}, p(x) = true\} \vdash^{I_c}_{I^b_c} \{\underline{p(x) = true}\} \vdash^{S_c}_{I^b_c} \{\underline{p(0) = true}, p(s(x')) = true\} \vdash^{D_c}_{I^b_c}$
$\{\underline{p(s(x')) = true}\} \vdash^{R_c}_{I^b_c} \{\underline{p(x') = true}, q(x', s(x')) = true\} \vdash^{I_c}_{I^b_c} \{\underline{q(x', s(x')) = true}\} \vdash^{I_c}_{I^b_c} \emptyset$,
where the processed conjectures are underlined.

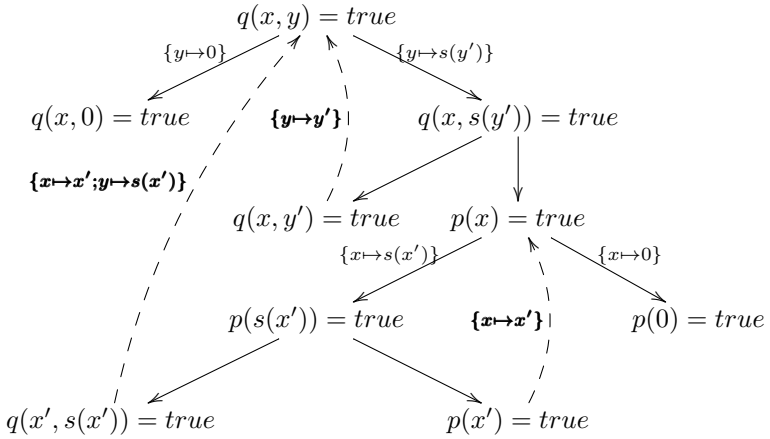

Fig. 2. The I^b_c-preproof of $\{q(x,y) = true\}$ as an oriented graph.

In order to check the soundness of the induction reasoning employed in D'-preproofs, we will illustrate the D'-preproofs as oriented graphs for which the nodes are equalities from the preproof and the arrows are of two kinds: i) *downward* arrows that link a processed conjecture to any new conjecture, and ii) *upward* (dashed) arrows that connect a processed conjecture to the previous conjecture whose instance was used as IH in an induction step. The instantiating substitutions used in split and induction steps annotate the corresponding arrows, those from the induction steps being written in boldface style. We denote a node by R *node* if R is the name of the D'-rule applied on the equality labelling the node. Also, an *IH-node* is any node labelled by an equality whose instance was used as IH.

Example 5.4. The I^b_c-preproof from Example 5.3 is illustrated as the oriented graph from Fig. 2.

A *cycle* of a D'-preproof can be represented as a circular list of paths such that each path has nodes from only one tree derivation of the preproof. Oriented graphs may have *minimal* cycles, i.e., cycles that do not contain other cycles, and *strongly connected components*. A strongly connected component p is a maximal sub-graph such that, given any two different nodes in p, there is a path in p between them in each direction.

Example 5.5. The oriented graph from Fig. 2 has 3 minimal cycles:
- $[q(x,y) = true, q(x, s(y')) = true, q(x,y') = true]$,
- $[p(x) = true, p(s(x')) = true, p(x') = true]$, and
- $[q(x,y) = true, q(x, s(y') = true, p(x) = true, p(s(x')) = true, q(x', s(x')) = true]$

and one strongly connected component representing the sub-graph built from the nodes of these minimal cycles. By abuse of notation, the nodes from the paths have been denoted by the labelling equalities. This is possible only for preproofs for which no equality labels distinct nodes in a tree derivation.

We can build a well-founded partial ordering $<_{\mathcal{C}}$ over the set of strongly connected components $\mathcal{C}$ of any D'-preproof. Given two strongly connected components p_1 and p_2, we write $p_1 <_{\mathcal{C}} p_2$ if there is a path in the cyclic graph of the D'-preproof leading any node of p_2 to any node of p_1.

5.2. Cyclic proofs

A D'-preproof of a multiset of equalities E^0, built using a set of axioms Ax, is *sound* if $Ax \models E^0$. Any sound preproof is also called a *proof*. In order to prove the soundness of a preproof, it is enough to show that any of its strongly connected components soundly uses the IHs.

Definition 5.6 (sound strongly connected components)**.** A strongly connected component p of a D'-preproof is *sound* if the IHs, representing instances of equalities labelling IH-nodes from p and used to process equalities labelling INDUCTION nodes from p, are true.

Ordering constraints that guarantee the soundness of D'-preproof can be defined for *normalised* cycles for which each path starts with an equality labelling a root of some tree derivation from the graph of p. This property can be achieved if any non-root IH-node is transformed into a root IH-node, as illustrated in Fig. 3 and referred to as the *normalisation operation*. Any transformation detaches the subtree rooted by the non-root IH-node, labelled by ϕ, from the graph to become a new tree, by preserving a copy of the IH-node. Next, an upward arrow is added to link the copy node with the root node of the new tree. By labelling it with the identity substitution σ^{ϕ}_{id}, the transformation simulates the application of INDUCTION on the formula labelling the copy node, using as IH-node the root node of the new tree. It can be noticed that the transformation does not generate new strongly connected components.

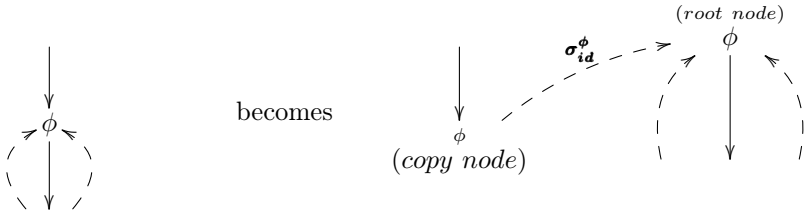

Fig. 3. The transformation of a non-root IH-node.

A strongly connected component is normalised if each of its minimal cycles is normalised. A D'-preproof is normalised if each of its strongly connected components is normalised. Since the number of non-root IH-nodes of a D'-preproof is finite, the normalisation process is finite. Moreover, the normal form is unique, independent from the order of processing the non-root IH-nodes.

Example 5.7. The I_c^b-preproof from Fig. 2 has one non-root IH-node, labelled by $p(x) = true$. The result of the transformation applied on it is illustrated in Fig. 4. The minimal cycles of the strongly connected component are all normalised:
- first cycle: $[q(x,u) = true, q(x,s(y')) = true, q(x,y') = true]$,
- second cycle: $[p(x) = true, p(s(x')) = true, p(x') = true]$, and
- third cycle: $[q(x,y) = true, q(x,s(y')) = true, p(x) = true]$, $[p(x) = true, p(s(x')) = true, q(x',s(x')) = true]$.

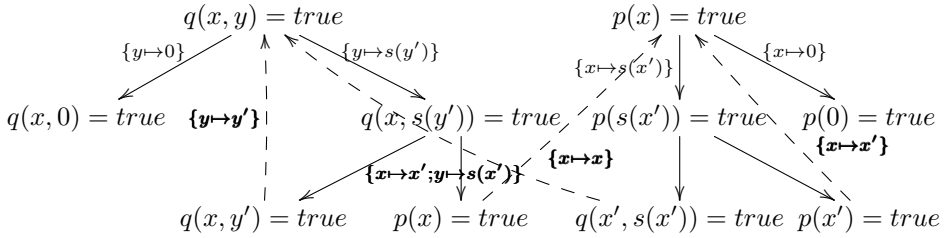

Fig. 4. The normalised I_c^b-preproof of $\{q(x,y) = true\}$.

Lemma 5.8. The normalisation of a D'-preproof of a multiset S of equalities is a new D'-preproof of a multiset S' of equalities such that $S \subseteq S'$.

Proof. The normalisation process may generate new tree derivations. If S'' is the set of the equalities labelling their root nodes, then S' is $S \cup S''$. □

Example 5.9. The I_c^b-preproof in Fig. 4 is also a I_c^b-preproof of $\{p(x) = true, q(x,y) = true\}$.

We can associate a substitution σ to each node n of a D'-preproof. If n is a direct offspring of a SPLIT node, then σ is the instantiating substitution used by the SPLIT operation to generate n, otherwise σ is the identity substitution. To each path in a tree derivation, of the form $[n_1, \ldots, n_k]$, we can also associate the *cumulative* substitution represented by the composition of substitutions $\S\sigma_1 \cdots \sigma_k$, where each σ_i $(i \in [1..k])$ is the substitution associated to the node n_i. We denote by $\phi(n)$ the equality labelling the node n.

Definition 5.10 (n-cycle discharging IHs)**.** An n-cycle, made of a circular list of $n(>0)$ paths $[n_1^1, \ldots n_1^{p1}], \ldots, [n_n^1, \ldots, n_n^{p_n}]$ from a strongly connected component p, *discharges* the IHs $\phi(n_j^1)\delta_j$ $(j \in [1..n])$ if, for any $i \in [1..n]$, we have that $\phi(n^1_{next(i)})\delta_{next(i)} <_p \phi(n_i^1)\theta_i$, where θ_i is the cumulative substitution for the path $[\phi(n_i^1), \ldots, \phi(n_i^{p_i})]$, $next(i) = 1 + (i \bmod n)$ and $<_p$ is a well-founded and 'stable under substitutions' ordering defined over the instances of the equalities labelling the root nodes of p.

Example 5.11. The IHs used in the minimal cycles from Fig. 4 and detailed in Example 5.7 are discharged if:
- $(q(x,y) = true)\{x \mapsto x; y \mapsto y'\} <_p (q(x,y) = true)\{x \mapsto x; y \mapsto s(y')\}$ in the first cycle,

- $(p(x) = true)\{x \mapsto x'\} <_p (p(x) = true)\{x \mapsto s(x')\}$ in the second cycle, and
- $(q(x,y) = true)\{x \mapsto x'; y \mapsto s(x')\} <_p (p(x) = true)\{x \mapsto s(x')\}$ and $(p(x) = true)\{x \mapsto x\} <_p (q(x,y) = true)\{x \mapsto x; y \mapsto s(y')\}$ in the third cycle,

where p denotes the strongly connected component of the I^b_c-preproof in Fig. 4. The ordering $<_p$ has to be defined such that the following four ordering constraints are satisfied: $q(x,y') = true <_p q(x,s(y')) = true$, $p(x') = true <_p p(s(x')) = true$, $q(x',s(x')) = true <_p p(s(x')) = true$ and $p(x) = true <_p q(x,s(y')) = true$. By proceeding similarly as in Section 4, we can define the measure value of an equality $l = r$ as the multiset of terms $|l| \cup |r|$, where $|t|$ is defined as

- $\{x, x\}$ if t is of the form $p(x)$,
- $\{x, x, y\}$ if t is of the form $q(x,y)$, and
- $\{t\}$, otherwise.

The ordering constraints are satisfied if the induction ordering used for comparing multisets of terms is the multiset extension of the rpo based on the precedence $false <_{\mathcal{F}} true <_{\mathcal{F}} 0 <_{\mathcal{F}} s$.

Theorem 5.12. [soundness of D'-preproofs] Any D'-preproof is sound if the minimal cycles from its normal form discharge their IHs.

Proof. The non-trivial induction reasoning is performed inside a strongly connected component. We say that the proof of an equality labelling a node n requires *non-trivial* induction reasoning if the sub-tree rooted by n has INDUCTION nodes whose corresponding IH-nodes are in the same strongly connected component as n. We will firstly prove the following soundness property of strongly connected components: the IHs discharged by the minimal cycles from any strongly connected component p of a normalised D' preproof are true if the other IHs used by the INDUCTION applications on nodes of p are true and the equalities from the direct offsprings of leaf nodes of p, if any, are true.

By contradiction, we assume that p has a false IH discharged by one of its minimal cycles. Since the number of minimal cycles in p is finite, we perform a classical induction reasoning on the number of minimal cycles discharging false IHs in p.

The base case. We assume that there is only one minimal cycle in p, denoted by $[n^1_1, \ldots n^{p1}_1], \ldots, [n^1_n, \ldots, n^{p_n}_n]$, that discharges the IHs $\phi(n^1_j)\delta_j$ ($j \in [1..n]$). So, for any $i \in [1..n]$, we have that $\phi(n^1_{next(i)})\delta_{next(i)} <_p \phi(n^1_i)\theta_i$, where θ_i is the cumulative substitution for the path $[n^1_i, \ldots, n^{p_i}_i]$, $next(i) = 1 + (i \bmod n)$ and $<_p$ is a well-founded and 'stable under substitutions' ordering defined over the instances of the equalities labelling the root nodes of p. W.l.o.g., we assume that the equality $\phi(n^1_n)$ has an instance used as a false IH in that minimal cycle.

Since $<_p$ is well-founded and no false instance of $\phi(n^1_n)$ can be proved outside the cycle, there is a $<_p$-minimal counterexample of $\phi(n^1_n)\theta_n$, denoted by $\phi(n^1_n)\theta_n\tau$. It can be noticed that $\phi(n^{p_n}_n)$ can also be generated by applying the split-free operations along the path $[n^1_n, \ldots, n^{p_n}_n]$. The equalities $\phi(n^1_n)\theta\tau, \ldots, \phi(n^{p_n}_n)\tau$ are all false because no extra minimal cycles using false IHs have been involved in any reasoning along the path $[n^1_n, \ldots, n^{p_n}_n]$. Moreover, the IH $\phi(n^1_1)\delta_1\tau$ should be false, as the new equalities resulting after the application of INDUCTION on $\phi(n^{p_n}_n)$ are all true. By the 'stability under substitutions' property of $<_p$ and the relation $\phi(n^1_1)\delta_1 <_p \phi(n^1_n)\theta_n$, we have that $\phi(n^1_1)\delta_1\tau <_p \phi(n^1_n)\theta_n\tau$. This also holds for any counterexample $\phi(n^1_1)\delta_1\tau\epsilon$ of $\phi(n^1_1)\delta_1\tau$

because $\phi(n_n^1)\theta_n\tau$ is ground, hence $\phi(n_1^1)\delta_1\tau\epsilon <_p \phi(n_n^1)\theta_n\tau$. In addition, $\phi(n_1^1)\delta_1\tau\epsilon$ cannot be proved outside the cycle, so it should be an instance of $\phi(n_1^1)\theta_1$. So there exists a substitution τ_1 such that $\phi(n_1^1)\delta_1\epsilon_1 \equiv \phi(n_1^1)\theta_1\tau_1$, where $\epsilon_1 \equiv \tau\epsilon$. Similarly, we show that there is a counterexample $\phi(n_2^1)\delta_2\epsilon_2$ smaller than $\phi(n_1^1)\theta_1\tau_1$. And so on, we end the visit of all the root nodes of the minimal cycle by building a counterexample $\phi(n_n^1)\theta_n\tau_n$ which is smaller than $\phi(n_{n-1}^1)\delta_{n-1}\epsilon_{n-1}$. By the transitivity of $<_p$, we derive the contradiction that $\phi(n_n^1)\theta_n\tau_n$ is a counterexample smaller than $\phi(n_n^1)\theta_n\tau$.

The step case. We consider the case when p has m (>1) minimal cycles that soundly discharge false IHs. By induction hypothesis, we assume that any subproof involving 'less than m' minimal cycles in p and soundly discharging false IHs is sound.

Let us assume that there is a minimal cycle in p, denoted by $[n_1^1, \ldots, n_1^{p1}], \ldots, [n_n^1, \ldots, n_n^{p_n}]$, that discharges the IHs $\phi(n_j^1)\delta_j$ ($j \in [1..n]$). It means that, for any $i \in [1..n]$, we have that $\phi(n_{next(i)}^1)\delta_{next(i)} <_p \phi(n_i^1)\theta_i$, where θ_i is the cumulative substitution for the path $[n_i^1, \ldots, n_i^{p_i}]$. As for the base case, we assume that the equality $\phi(n_n^1)$ has an instance used as a false IH in the cycle. Moreover, it is an instance of $\phi(n_n^1)\theta_n$, otherwise it should be true since its proof involves 'less than m' minimal cycles soundly discharging false IHs, as assumed by induction hypothesis.

We denote by $\phi(n_n^1)\theta_n\tau$ a $<_p$-minimal counterexample of $\phi(n_n^1)\theta_n$. $\phi(n_n^{p_n})\tau$ is also a counterexample because the IHs used during the INDUCTION steps along the path $[n_n^1, \ldots, n_n^{p_n}]$ are true, as their subproofs involve 'less than m' minimal cycles soundly discharging false IHs. Moreover, $\phi(n_1^1)\delta_1\tau$ is false because all the direct offsprings of $n_n^{p_n}$ are labelled with true equalities requiring 'less than m' minimal cycles soundly discharging false IHs. By the 'stability under substitutions' property of $<_p$ and the relation $\phi(n_1^1)\delta_1 <_p \phi(n_n^1)\theta_n$, we have that $\phi(n_1^1)\delta_1\tau <_p \phi(n_n^1)\theta_n\tau$. As for the base, there is a counterexample $\phi(n_1^1)\delta_1\epsilon_1$ of $\phi(n_1^1)\delta_1\tau$ such that $\phi(n_1^1)\delta_1\epsilon_1 <_p \phi(n_n^1)\theta_n\tau$. $\phi(n_1^1)\delta_1\epsilon_1$ is also a counterexample of $\phi(n_1^1)\theta_1$, so it can be treated similarly as $\phi(n_n^1)\theta_n\tau$. By traversing all the nodes in the cycle, we can build a strictly decreasing chain of counterexamples of equalities labelling the root nodes of the cycle, starting with $\phi(n_n^1)\theta_n\tau$ and ending with a counterexample of $\phi(n_n^1)\theta_n$, contradicting the $<_p$-minimality property of $\phi(n_n^1)\theta_n\tau$.

This ends the proof of the property.

Next, we assume by contradiction that the minimal cycles of the normal form of a D'-preproof discharge their IHs but there is an equality proved by the D'-preproof which is false. By Lemma 5.8, this equality labels one of the root nodes of the normal form of the D'-preproof.

Let $\mathcal{C}$ denote the set of strongly connected components of the normalised D'-preproof, $<_\mathcal{C}$ the well-founded ordering over $\mathcal{C}$, and assume that there is a strongly connected component p in $\mathcal{C}$ such that the false equality labels one of its root nodes, denoted by n. We perform a classical induction reasoning on the elements of $\mathcal{C}$.

The base case. p is a $<_\mathcal{C}$-minimal strongly connected component for which the equalities from the direct offsprings of leaf nodes of p are true since no induction reasoning is required in their subproofs. The IHs used by the INDUCTION applications on nodes of p and representing instances of equalities labelling nodes outside p are

true because their proofs do not require non-trivial induction reasoning, thanks to the $<_{\mathcal{C}}$-minimality of p. Therefore, the above property can be applied to conclude that all IHs used in the tree derivation rooted by n are true, which means that equality labelling n is true, so contradiction.

The step case. Let n be the root of some strongly connected component p from $\mathcal{C}$ and, by induction hypothesis, we assume that all the root nodes of any strongly connected component of $\mathcal{C}$, $<_{\mathcal{C}}$-smaller than p, are labelled by true equalities. The IHs used by the INDUCTION applications on nodes of p and representing instances of equalities labelling nodes outside p are true because they are instances of equalities labelling either i) root nodes of strongly connected components of $\mathcal{C}$ $<_{\mathcal{C}}$-smaller than p, or ii) root nodes of tree derivations that are not in any strongly connected component of $\mathcal{C}$ but whose proofs may require non-trivial induction reasoning captured only inside strongly connected components $<_{\mathcal{C}}$-smaller than p. From the above property, all IHs used in the tree derivation rooted by n are true. Hence, the equality labelling n is true, which leads to a contradiction.

We conclude that all root nodes of any strongly connected component of $\mathcal{C}$ are labelled with true equalities.

Therefore, n is a root node that is not the member of any strongly connected component from $\mathcal{C}$. We can build an ordering $<_{\mathcal{R}}$ on the set $\mathcal{R}$ of root nodes that are not part of any strongly connected component from $\mathcal{C}$, as follows: $n_1 <_R n_2$ if there is an INDUCTION step in the tree rooted by n_2 for which n_1 is the IH-node. The $<_{\mathcal{R}}$ ordering is well-founded, otherwise it would include a cycle which contradicts the fact that the nodes from $\mathcal{R}$ are not members of any strongly connected component of $\mathcal{C}$. We reason by induction on the nodes of $\mathcal{R}$. If n is a $<_{\mathcal{R}}$-minimal node, then its equality is true since the IHs used in the tree derivation rooted by n, if any, are true, as instances of equalities labelling root nodes from $\mathcal{C}$. If n is not a $<_{\mathcal{R}}$-minimal node, then we assume by induction hypothesis that all nodes from $\mathcal{R}$, $<_{\mathcal{R}}$-smaller than n, are labelled by true equalities. The equality labelling n is true because the IHs used in the tree derivation rooted by n are true, as instances of equalities labelling root nodes from $\mathcal{C}$ or nodes from $\mathcal{R}$, $<_{\mathcal{R}}$-smaller than n. This leads again to a contradiction. □

Example 5.13 (cont. Example 5.11)**.** The I_c^b-preproof from Fig. 4 is sound because all the IHs from its minimal cycles are discharged. Therefore, $q(x, y) = true$ and $p(x) = true$ are true.

5.3. Coq formalisation and certification of cyclic proofs using the logic programming style

The p and q functions can be defined using the inductive predicates **P** and **Q**, respectively, by following the logic programming style:

```
Inductive P : nat → Prop :=
  | p0 : P 0
  | p1 : ∀ x : nat, P x → Q x (S x) → P (S x)
with Q : nat → nat → Prop :=
```

```
| q0 : ∀ x, Q x 0
| q1 : ∀ x y : nat, Q x y → P x → Q x (S y).
```

Any equality of the form $t = true$ is translated to the atom t. The variables from the equational specifications and cyclic proofs are either universally quantified or free variables in the corresponding Coq specifications and proofs.

The formula-based Noetherian induction principle will be applied on the set built only from the equalities whose instances are used as induction hypotheses in the cycles, in our case, the two atoms labelling the root nodes. The LF list LF_PandQ and its type type_LF_PandQ are:

```
Definition type_LF_PandQ := nat → nat → Prop × List.list term.
Definition LF_PandQ:= [Pu, Qxy].
```

where the anonymous functions Pu and Qxy associate to the root atoms the measure values defined at Example 5.11:

```
Definition Pu : type_LF_PandQ :=
  fun u _ ⇒ (P u, (model_nat u :: model_nat u :: nil)).
Definition Qxy : type_LF_PandQ :=
  fun x y ⇒ (Q x y, (model_nat x :: model_nat x :: model_nat y :: nil)).
```

The main lemma becomes:

```
Lemma main_PandQ : ∀ F, In F LF_PandQ → ∀ u, (∀ F', In F' LF_PandQ → ∀ u', less
(snd (F' u')) (snd (F u)) → fst (F' u')) → fst (F u).
```

Its proof starts by a case analysis on the possible values of F, which can be either Pu or Qxy. The generation of the Coq script for each case starts by a split rule instantiating a natural variable with 0 and the successor of a fresh variable. In Coq, this can be easily performed with the `destruct` tactic or by the means of *functional schemes*, as shown in the proof of Lemma main_13 at the end of Section 4.

The proof of each resulted case follows some path from a root node to a leaf node in the cyclic graph from Figure 4. The generation of the Coq script can be automatised since there is a direct Coq translation of the non-split inference rules, as follows:

- DelInst' is translated to the `apply` tactic parameterised by the name of the used axiom;
- RedEq' is translated by unfolding the definition of the processed conjecture, using again the `apply` tactic parameterised by the name of the used axiom;
- the application of IndNat using an induction hypothesis from LF_PandQ is translated to

```
pose proof (HFabs0 F) as Hind. clear HFabs0.
assert (fst (F _u1 0)) as HFabs0.
apply Hind. trivial_in n.
```

The user-defined tactic *trivial_in* is applied on an index n and checks that F is the $(n+1)$ element of the LF_PandQ list.

Next, any formula defining the anonymous functions of LF_PandQ is proved:

```
Theorem all_true_PandQ: ∀ F, In F LF_PandQ → ∀ u1: nat, fst (F u1).
```

Finally, $\forall u, P(u)$ and $\forall x\ y, Q(x,y)$ are certified by the following theorems:

```
Theorem true_Pu : ∀ u : nat, P u.
```

```
Theorem true_Qxy : ∀ (x : nat) (y : nat), Q x y.
```

The details of the Coq proof can be found in Appendix A.1.

5.4. Coq formalisation and certification of cyclic proofs using the functional programming style

Conjecture (19) can also be proved using the cyclic inference system I_c^f, based on the reasoning techniques underlying I^f, presented in Section 4:

SplitNat' (S'_c): $E \cup \{\phi\langle x\rangle\} \vdash_{I_c^f} E \cup \{\phi\{x \mapsto 0\}, \phi\{x \mapsto s(x')\}\}$,
where x' is a fresh variable.

DelTaut (T_c): $E \cup \{\phi\} \vdash_{I_c^f} E$,
if ϕ is a tautology.

NegClash (N_c): $E \cup \{\phi\} \vdash_{I_c^f} E$
if $true = false$ or $false = true$ is in the condition part of ϕ.

RedAx (A_c): $E \cup \{\phi\} \vdash_{I_c^f} E \cup \{\psi\}$,
if $\phi \rightarrow_{Ax} \psi$.

DelSub (E_c): $E \cup \{\phi\} \vdash_{I_c^f} E$
if ϕ is an instance of a previously generated equality.

The SplitNat' rule is similar to the I_c^b-rule SplitNat. DelTaut and NegClash delete tautologies and conditional equalities with false conditions, respectively. RedAx rewrites the processed conjecture with axioms. Finally, DelSub deletes the processed conjecture if it is an instance of a previous equality from the I_c^f-preproof.

Theorem 5.14. Every I_c^f-rule instantiates a D'-rule.

Proof. We perform a case analysis on the I_c^f-rules:
- SplitNat' is an instance of Split, for the same reasons given for SplitNat in the proof of Theorem 5.2;
- DelTaut and NegClash are instances of Deduction when the set of new conjectures is empty. This situation is acceptable because the processed conjectures are valid;

- RedAx is an instance of Deduction because for any counterexample $\phi\tau$ of the processed conjecture ϕ, $\psi\tau$ is a counterexample of the new conjecture ψ;
- DelSub is an instance of Induction for similar reasons given for IndNat in the proof of Theorem 5.2.

□

In the following, we denote by GenNat' the derived I_c^f-rule that abbreviates the application of SplitNat' rule on the processed conjecture, followed by the application of RedAx on each of the two new conjectures:

GenNat' (G_c): $E \cup \{\phi\langle x\rangle\} \vdash_{I_c^f} E \cup \{\phi_1, \phi_2\}$,
where $\phi\{x \mapsto 0\} \rightarrow_{Ax} \phi_1$, $\phi\{x \mapsto s(x')\} \rightarrow_{Ax} \phi_2$ and x' is a fresh variable.

By defining the atoms:

$e'_{13}(u_1, u_2, u_3) : odd(u_1 + u_2) = true \wedge even(u_2 + u_3) = true \Rightarrow odd(u_1 + u_3) = true$
$e'_{23}(u_2, u_3) : odd(0 + u_2) = true \wedge even(u_2 + u_3) = true \Rightarrow odd(u_3) = true$
$e'_{29}(u_4, u_2, u_3) : odd((s(u_4)) + u_2) = true \wedge even(u_2 + u_3) = true \Rightarrow odd(s(u_4 + u_3)) = true$
$e'_{36}(u_2, u_3) : odd(u_2) = true \wedge even(u_2 + u_3) = true \Rightarrow odd(u_3) = true$
$e'_{49}(u_4, u_2, u_3) : even(u_4 + u_2) = true \wedge even(u_2 + u_3) = true \Rightarrow even(u_4 + u_3) = true$
$e'_{67}(u_3) : odd(0) = true \wedge even(u_3) = true \Rightarrow odd(u_3) = true$
$e'_{73}(u_4, u_3) : odd(s(u_4)) = true \wedge even(s(u_4 + u_3)) = true \Rightarrow odd(u_3) = true$
$e'_{81}(u_3) : false = true \wedge even(u_3) = true \Rightarrow odd(u_3) = true$
$e'_{91}(u_4, u_3) : even(u_4) = true \wedge odd(u_4 + u_3) = true \Rightarrow odd(u_3) = true$
$e'_{113}(u_2, u_3) : even(0 + u_2) = true \wedge even(u_2 + u_3) = true \Rightarrow even(u_3) = true$
$e'_{119}(u_5, u_2, u_3) : even((s(u_5)) + u_2) = true \wedge even(u_2 + u_3) = true \Rightarrow even(s(u_5 + u_3)) = true$
$e'_{138}(u_2, u_3) : even(u_2) = true \wedge even(u_2 + u_3) = true \Rightarrow even(u_3) = true$
$e'_{197}(u_3) : even(0) = true \wedge odd(u_3) = true \Rightarrow odd(u_3) = true$
$e'_{203}(u_5, u_3) : even(s(u_5)) = true \wedge odd(s(u_5 + u_3)) = true \Rightarrow odd(u_3) = true$
$e'_{253}(u_3) : even(0) = true \wedge even(u_3) = true \Rightarrow even(u_3) = true$
$e'_{259}(u_5, u_3) : even(s(u_5)) = true \wedge even(s(u_5 + u_3)) = true \Rightarrow even(u_3) = true$
$e'_{283}(u_5, u_3) : odd(u_5) = true \wedge odd(u_5 + u_3) = true \Rightarrow even(u_3) = true$
$e'_{311}(u_3) : odd(0) = true \wedge odd(u_3) = true \Rightarrow even(u_3) = true$
$e'_{317}(u_6, u_3) : odd(s(u_6)) = true \wedge odd(s(u_6 + u_3)) = true \Rightarrow even(u_3) = true$
$e'_{333}(u_3) : false = true \wedge odd(u_3) = true \Rightarrow even(u_3) = true$

one can build the following I_c^f-preproof of the conjecture (19), represented as a linear derivation:

$\{\underline{e'_{13}(u_1, u_2, u_3)}\} \vdash^{G_c}_{I_c^f} \{\underline{e'_{23}(u_2, u_3)}, e'_{29}(u_4, u_2, u_3)\} \vdash^{A_c}_{I_c^f} \{e'_{36}(u_2, u_3), \underline{e'_{29}(u_4, u_2, u_3)}\} \vdash^{*A_c}_{I_c^f}$
$\{\underline{e'_{36}(u_2, u_3)}, e'_{49}(u_4, u_2, u_3)\} \vdash^{G_c}_{I_c^f} \{\underline{e'_{67}(u_3)}, e'_{73}(u_4, u_3), e'_{49}(u_4, u_2, u_3)\} \vdash^{A_c}_{I_c^f}$
$\{\underline{e'_{81}(u_3)}, e'_{73}(u_4, u_3), e'_{49}(u_4, u_2, u_3)\} \vdash^{N_c}_{I_c^f} \{\underline{e'_{73}(u_4, u_3)}, e'_{49}(u_4, u_2, u_3)\} \vdash^{*A_c}_{I_c^f}$

$\{\underline{e'_{91}(u_4,u_3)}, e'_{49}(u_4,u_2,u_3)\} \quad \vdash^{G_c}_{I_c^f} \quad \{\underline{e'_{197}(u_3)}, e'_{203}(u_5,u_3), e'_{49}(u_4,u_2,u_3)\} \quad \vdash^{T_c}_{I_c^f}$
$\{\underline{e'_{203}(u_5,u_3)}, e'_{49}(u_4,u_2,u_3)\} \vdash^{*A_c}_{I_c^f} \{\underline{e'_{36}(u_5,u_3)}, e'_{49}(u_4,u_2,u_3)\} \vdash^{E_c}_{I_c^f} \{\underline{e'_{49}(u_4,u_2,u_3)}\} \vdash^{G_c}_{I_c^f}$
$\{e'_{113}(u_2,u_3), \underline{e'_{119}(u_5,u_2,u_3)}\} \vdash^{*A_c}_{I_c^f} \{e'_{113}(u_2,u_3), \underline{e'_{13}(u_5,u_2,u_3)}\} \vdash^{E_c}_{I_c^f} \{\underline{e'_{113}(u_2,u_3)}\} \vdash^{A_c}_{I_c^f}$
$\{\underline{e'_{138}(u_2,u_3)}\} \vdash^{G_c}_{I_c^f} \{\underline{e'_{253}(u_3)}, e'_{259}(u_5,u_3)\} \vdash^{T_c}_{I_c^f} \{\underline{e'_{259}(u_5,u_3)}\} \vdash^{*A_c}_{I_c^f} \{\underline{e'_{283}(u_5,u_3)}\} \vdash^{G_c}_{I_c^f}$
$\{\underline{e'_{311}(u_3)}, e'_{317}(u_6,u_3)\} \quad \vdash^{A_c}_{I_c^f} \quad \{\underline{e'_{333}(u_3)}, e'_{317}(u_6,u_3)\} \quad \vdash^{N_c}_{I_c^f} \quad \{\underline{e'_{317}(u_6,u_3)}\} \quad \vdash^{*A_c}_{I_c^f}$
$\{\underline{e'_{138}(u_6,u_3)}\} \vdash^{E_c}_{I_c^f} \emptyset$

Fig. 5 illustrates the above preproof as an oriented graph. We can distinguish three strongly connected components, denoted by p_1, p_2 and p_3, each of them made of only one minimal cycle, as follows:

- $[e'_{13}(u_1,u_2,u_3), e'_{29}(u_4,u_2,u_3), e'_{49}(u_4,u_2,u_3), e'_{119}(u_5,u_2,u_3), e'_{13}(u_5,u_2,u_3)]$ for p_1,
- $[e'_{36}(u_2,u_3), e'_{73}(u_4,u_3), e'_{91}(u_4,u_3), e'_{203}(u_5,u_3), e'_{36}(u_5,u_3)]$ for p_2, and
- $[e'_{138}(u_2,u_3), e'_{259}(u_5,u_3), e'_{283}(u_5,u_3), e'_{317}(u_6,u_3), e'_{138}(u_6,u_3)]$ for p_3.

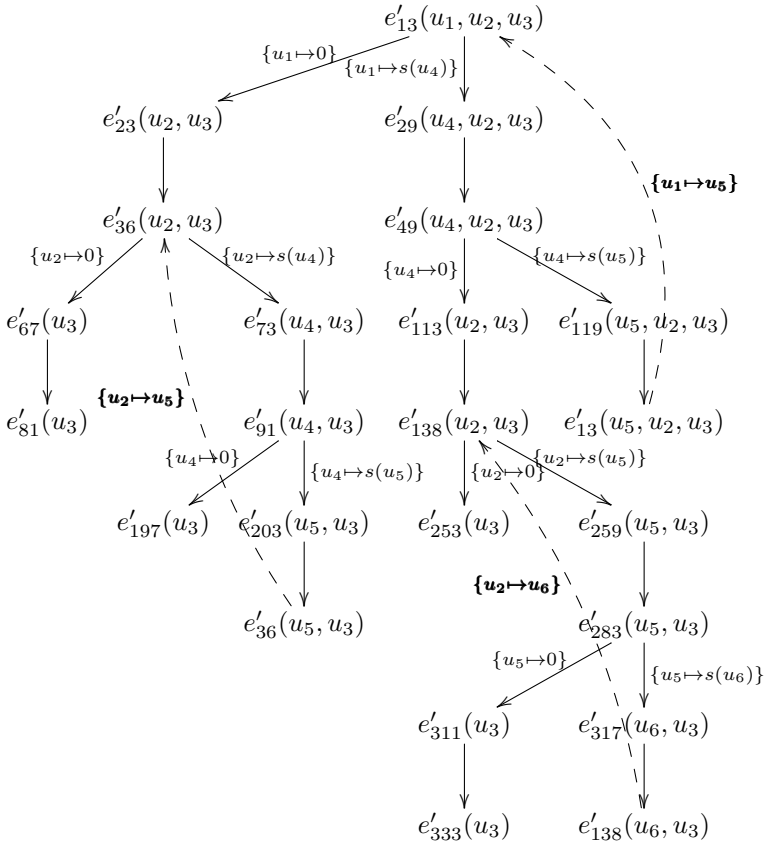

Fig. 5. The graph representation of the I_c^f-preproof of $\{e'_{13}(u_1,u_2,u_3)\}$.

The normalisation process of the I_c^f-preproof applies the transformation from Fig. 3 to the two non-root IH-nodes labelled by $e'_{36}(u_2,u_3)$ and $e'_{138}(u_2,u_3)$. The normal form

of the I_c^f-preproof consists in three derivation trees, rooted by the nodes labelled by $e'_{13}(u_1, u_2, u_3)$, $e'_{36}(u_2, u_3)$ and $e'_{138}(u_2, u_3)$.

The soundness of the I_c^f-preproof is ensured if the following three constraints are satisfied: $e'_{13}(u_1, u_2, u_3)\{u_1 \mapsto s(s(u_5)); u_2 \mapsto u_2; u_3 \mapsto u_3\} <_{p_1} e'_{13}(u_1, u_2, u_3)\{u_1 \mapsto u_5; u_2 \mapsto u_2; u_3 \mapsto u_3\}$, $e'_{36}(u_2, u_3)\{u_2 \mapsto u_5; u_3 \mapsto u_3\} <_{p_2} e'_{36}(u_2, u_3)\{u_2 \mapsto s(s(u_5)); u_3 \mapsto u_3\}$, and $e'_{138}(u_2, u_3)\{u_2 \mapsto u_6; u_3 \mapsto u_3\} <_{p_3} e'_{138}(u_2, u_3)\{u_2 \mapsto s(s(u_6)); u_3 \mapsto u_3\}$. Different well-founded and 'stable under substitutions' orderings over multisets of terms can be used to implement $<_{p_1}$, $<_{p_2}$ and $<_{p_3}$, for example the multiset extension of the rpo using any precedence.

The *strategy* for certifying a normalised D'-preproof integrating a (potentially empty) set $\mathcal{C}$ of strongly connected components is based on a partial ordering defined on the components of some partition P over the root nodes of the normalised D'-preproof. P is built such that two nodes are in the same component if they belong to the same strongly connected component of $\mathcal{C}$. The partial ordering, denoted by $<^P$, extends $<_{\mathcal{C}}$ in such a way that, for any two components c_1 and c_2, we have $c_1 <^P c_2$ if there is an INDUCTION node in the trees rooted by nodes from c_2 whose corresponding IH-node is a root node from c_1. The strategy is to certify the proofs of the multisets of equalities labelling the root nodes of each component from P, in distinctive steps and in increasing ordering w.r.t. $<^P$.

Example 5.15. The certification of the normalised I_c^f-preproof of $\{e'_{13}(u_1, u_2, u_3)\}$ starts with the certification of the proofs of the equalities labelling the root nodes of the $<_{\mathcal{C}}$-minimal strongly connected components, i.e., p_2 and p_3, followed by the certification of the proof of the equality labelling the root node of p_1.

The certification process for the proof of the multiset of equalities labelling the root nodes of a component c of P is similar to that given for implicit induction proofs in Subsection 4.4. First, we define the LF list as the set of anonymous functions built for the equalities labelling the root nodes from c. Second, the instantiation schemas from the GENNAT' steps are defined using functional schemes, or the `destruct` tactic. Third, the proof of the main lemma is built by translating in Coq script the I_c^f-steps encountered by following the paths from root to leaf nodes in the trees rooted by the nodes of c, using similar translations as shown in Subsection 4.4. In addition, the application as IH of any instance of some equation e'_n not labelling any node from c is translated to `apply` *true_n*. The proposed certification strategy ensures that the proof of the theorem *true_n* is certified before its use by the `apply` tactic. Multiple rewrite steps can be performed with the `simpl` tactic. Finally, the theorem all_true is certified, followed by the certification of the *true* theorems about the equalities labelling each node of c.

As example, we will only detail the certification process for the proof of $e'_{13}(u_1, u_2, u_3)$, knowing that the certification of the proofs for $e'_{36}(u_2, u_3)$ and $e'_{138}(u_2, u_3)$ was already completed and done in a similar way. The LF list for p_1 is defined as:

`Definition` *LF_13* := [*F_13*].

where *F_13* is defined as F_13 from Subsection 4.4.

The functional schemes associated to the GENNAT' steps are defined as:

```
Fixpoint f_13 (u1 : nat) (u3 : nat) {struct u1}: nat :=
  match u1, u3 with
  | 0, _ ⇒ 0
  | S u4, _ ⇒ 0
  end.

Functional Scheme f_13_ind := Induction for f_13 Sort Prop.

Fixpoint f_49 (u4 : nat) (u3 : nat) {struct u4}: nat :=
  match u4, u3 with
  | 0, _ ⇒ 0
  | S u5, _ ⇒ 0
  end.

Functional Scheme f_49_ind := Induction for f_49 Sort Prop.
```

The Coq script of the *main_13* lemma is:

```
Lemma main_13 : ∀ F, In F LF_13 → ∀ u1 u2 u3, (∀ F', In F' LF_13 → ∀ e1 e2 e3,
  less (snd (F' e1 e2 e3)) (snd (F u1 u2 u3)) → fst (F' e1 e2 e3)) → fst (F u1 u2
u3).
Proof.
intros F HF u1 u2 u3; case_In HF; intro Hind.

(* GenNat' on e'_13  *)

rename u1 into _u1. rename u2 into _u2. rename u3 into _u3.
rename _u1 into u1. rename _u2 into u2. rename _u3 into u3.

revert Hind.

pattern u1, u3, (f_13 u1 u3). apply f_13_ind.

(* case e'_23  *)

intros _u1 _u3. intro. intro HFabs0.
simpl. apply true_36.

(* case e'_29  *)

intros _u1 _u3. intro u4. intro. intro HFabs0.
simpl.

(* GenNat' on e'_49  *)

destruct u4. (* we could have used instead the functional scheme f_49_ind  *)

(* case e'_113  *)

simpl. apply true_138.

(* case e'_119  *)

pose proof (HFabs0 F_13) as Hind. clear HFabs0.
assert (fst (F_13 u4 u2 _u3)) as HFabs0.
apply Hind. trivial_in 0.
unfold snd. unfold F_13. rewrite_model. abstract solve_rpo_mul.
simpl. simpl in HFabs0. trivial.
Qed.
```

The Coq script for the proofs of the remaining theorems is omitted, but it is available online as supplementary material.

```
Theorem all_true_13 :
 ∀ F,
 In F LF_13 → ∀ (u1 : nat) (u2 : nat) (u3 : nat), fst (F u1 u2 u3).

Theorem true_13 :
 ∀ (u1 : nat) (u2 : nat) (u3 : nat),
 odd (plus u1 u2) = true →
 even (plus u2 u3) = true → odd (plus u1 u3) = true.
```

6. Automatic certification of SPIKE proofs

General presentation of SPIKE The automatic theorem prover SPIKE was created in a period when several formula-based Noetherian induction methods issued from Musser's completion-based inductionless induction (or proof-by-consistency) technique (Musser, 1980) have been devised. Some of them have been implemented into theorem provers, for example, the RRL (Kapur and Zhang, 1988) and Focus (Bronsard and Reddy, 1991) systems integrated the test-set induction method (Kapur et al., 1986) and a generalization of the term-rewriting induction (Bronsard et al., 1994) for conditional theories, respectively. Inspired by the rewriting techniques previously tested with the ORME system (Lescanne, 1990), SPIKE (Bouhoula et al., 1992) implemented an implicit induction method (Kounalis and Rusinowitch, 1990; Bouhoula et al., 1995) that combines features from explicit induction and inductive completion techniques. It mainly served as a prototype for proving conjectures about (extensions of) conditional specifications and for testing induction-based reasoning techniques that led to many successful proof experiments on non-trivial applications.

SPIKE can reason on conditional specifications that should satisfy some crucial properties, like the *ground convergence* and *completeness*, that ensure the coherence of the axioms and the validity of the variable instantiation schemas, respectively. They can be checked more easily if the specification is many-sorted and the set of function symbols is split into constructors and defined function symbols. SPIKE was initially designed to deal with free constructors for which no equality relation can be established between any two different constructor symbols. Several extensions have been operated on SPIKE since (Bouhoula et al., 1992) in order to deal with: i) non-free constructors (Bouhoula and Jouannaud, 2001), ii) parameterised specifications (Bouhoula, 1994, 1996), iii) associative-commutative theories (Berregeb et al., 1996), iv) observational proofs (Berregeb et al., 1998; Bouhoula and Rusinowitch, 2002), and v) simultaneous check of the completeness and ground convergence properties of a specification (Bouhoula, 2009). Most of them led to distinct proof systems that are no longer maintained in spite of their theoretical and practical interests.

The inference system Each inference rule is the implementation of one of the three rules of the abstract inference system A, based on contextual cover sets (CSSs) (Stratulat, 2001): AddPremise, Simplify, and Delete. The first two rules are the representation with CCSs of the corresponding A'-rules from Section 4.1. The last rule is a particular

case of SIMPLIFY that generates an empty set of new conjectures. Thanks to a strategy language (Alouini and Bouhoula, 1997), new rules can be defined by the user to combine existing reasoning techniques. It also controls the way the proofs are built and allows the prover to automatically generate large proofs and to deal with non-trivial applications. Other reasoning techniques are implemented in built-in rules that are activated without the help of the strategy language, e.g., the combination between a decision procedure for linear arithmetic and a congruence closure algorithm (Stratulat, 2000; Armando et al., 2002; Stratulat, 2014). The arithmetic reasoning permitted to validate the MJRTY algorithm (Boyer and Moore, 1991) using a lemma proposed by N. Shankar (according to (Howe, 1993)); also, more than 60% of the conjectures required to certify the conformity algorithm for a telecommunication protocol (Rusinowitch et al., 2003) have been automatically proved.

The validation of the JavaCard platform (Barthe and Stratulat, 2003) was the most challenging case study ever experienced by SPIKE. The inference system has been adapted to manage variables of parameterised sorts as well as existential variables. New inference rules have been designed to better handle the information from the conditional part of (conditional) conjectures. The efficiency of the implementation has been improved for dealing with specifications counting more than 400 defined function symbols and 2200 axioms, for example by recording the failure context at the conjecture level in order to avoid useless computation.

Recently (Stratulat, 2012), the inference system of SPIKE has been extended to implement reductive-free cyclic proofs by keeping the best features of explicit and implicit induction reasoning. The induction reasoning may not be rewrite-based and axioms are no longer required to be oriented into rewrite rules, hence allowing for more general specifications. The prover integrates the DRaCuLa strategy to build cycles by need. Mainly, the induction steps from a cycle are blocked until any involved IH is either proved or discharged by cycles. When a cycle is built, its induction steps are unblocked if they are discharged by the cycle, allowing for *simultaneous* induction.

Layout of a specification file The structure of a standard SPIKE specification, included in a file *name*`.spike`, is:

`specification:` *name*
% the axiomatic definition of a many-sorted constructor-based specification
`sorts:` *list of sorts*
`constructors:` *list of constructor symbols*
`defined functions:` *list of defined function symbols*
`axioms:` *list of axioms for each defined function symbols*
% the induction ordering
`greater:` *list of precedences over the function symbols*
`equiv:` *list of equivalent function symbols*
% the completeness and ground convergence properties
`properties:` *list of properties*
% the proof strategies
`strategy:` *list of inference rules and proof strategies*
% the conjectures to be proved
`conjectures:` *list of conjectures*

The employed induction ordering is the multiset extension of the mpo ordering (Baader and Nipkow, 1998) using the precedence over the function symbols defined at the sections `greater` and `equiv` from the specification file. The mpo ordering also serves to orient the axioms into rewrite rules.

In Appendix A.2, we provide the SPIKE specification for the *even* and *odd* example from Section 4.

Proving properties The implicit induction proof for a conjecture from a specification file *name*.`spike` is generated by the command `spike_bc` *name*.`spike`. Its generation is highly automatic, following a *push-button* approach, but the user may influence the process of proof construction before launching the proof by defining i) the precedence used by the mpo ordering, ii) the inference rules and the proof strategy, iii) the precedence over the head symbols of the (sub)terms to which the new instantiation technique can be applied, and iv) lemmas. Once a conjecture has been proved, it can participate as lemma in the proof of further conjectures listed in the `conjectures` section.

The user can also interact with the prover by the means of i) extra sections, for example `use: nats;` [2] for activating the combination of the decision procedure for linear arithmetic and the congruence closure procedure, and ii) command-line arguments given to `spike_bc`. Some of the useful arguments are:

`-coqc_spec`: generate the preambule, including the definition of the ordering and the Coq specification, and save the result in the Coq file *name*`_spec.v`;
`-coqc`: translate the implicit induction proof and save the result in the Coq file *name*`.v`;
`-dracula`: generate cyclic proofs using the DRaCuLa strategy.

In an automatic way, SPIKE can prove the conjecture (19) from Section 4 by both implicit and cyclic induction. The numerical annotations of the atoms from the presented proofs correspond to numbers labelling conjectures in the SPIKE proofs. SPIKE can also automatically translate the implicit induction proof into valid Coq script. In order to do this, the Coq script translating the axioms and model functions is inlined in the SPIKE specification and should be provided by the user. It is prefixed by $ and ignored by SPIKE during the proof development. On the other hand, the cyclic proof was manually translated into Coq script by modifying the Coq script generated for the implicit induction proof.

7. Conclusions and future work

We have provided the formal tools to certify formula-based Noetherian induction reasoning with Coq. As an alternative to the built-in explicit induction techniques, we have opened the perspective to directly implement in Coq formula-based Noetherian induction methods that effectively manage the lazy, simultaneous and mutual induction reasoning. Compared to the methods consisting in translating particular classes of formula-based Noetherian induction proofs to an explicit induction form, our approach can generate a

[2] To be added just after the `specification` section.

constructive Coq proof from *any* formula-based Noetherian induction proof. The main challenges to face are i) the explicit representation of the underlying induction ordering that is not built-in in Coq and that should be supported by external libraries, and ii) the automatisation of the certification process. Classical Coq proofs can also be built using the 'Descente Infinie' induction principle (Stratulat, 2010).

The formal tools have been used to automatically certify implicit induction and cyclic proofs. The implicit induction reasoning is reductive and can be easily automatised, while the cyclic reasoning is reductive-free, requires fewer ordering constraints and allows for more general specifications, but is less automatisable (Stratulat, 2012). In practice, the Coq script translating cyclic proofs may be (much) shorter than that for implicit induction proofs, hence easier to certify. The number of ordering constraints and the size of the LF lists has a strong impact on the complexity of the generated Coq script. For instance, the script generated from the implicit induction proof of the conjecture (19) deals with a unique LF list of 28 anonymous functions and certifies 30 ordering constraints, while the Coq script translating the cyclic proof has only three singleton LF lists and 3 ordering constraints to be checked.

The certification methodology has been tested with SPIKE on different other examples, among which the validation of a conformity algorithm for a telecommunication protocol (Rusinowitch et al., 2003). As shown in (Stratulat and Demange, 2011), most of the lemmas have been automatically certified by Coq. On the other hand, the methodology is limited for several reasons. SPIKE cannot translate the proof steps built with some of its inference rules, e.g., those requiring arithmetic reasoning due to the complexity of the underlying decision procedures, or some rules are correctly translated only under certain conditions. In fact, the translation process does not guarantee the conversion of any SPIKE proof to a valid Coq script. Also, an inconvenient for the potential users is represented by the necessity to inline Coq code in the SPIKE specification.

In a future line of research, we intend to avoid the translation drawbacks by directly generating the formula-based Noetherian induction proofs with the Coq inference system. In another direction, we plan to apply the formal tools for certifying other formula-based Noetherian induction techniques, e.g., the inductionless induction reasoning and other saturation-based reductive reasoning, as those based on ordered paramodulation (Nieuwenhuis and Rubio, 2001) and resolution (Bachmair and Ganzinger, 2001). Last but not least, we intend to build similar formal tools for other certifying proof assistants, like Isabelle.

Acknowledgements The author thanks Vincent Demange and Amira Henaien for useful discussions as well as the anonymous referees for the suggestions and comments that helped to improve the quality of the paper.

References

Alouini, I., Bouhoula, A., 1997. Un langage de stratégie pour SPIKE. Working document.

Aoto, T., 2006. Dealing with non-orientable equations in rewriting induction. In: Pfenning, F. (Ed.), RTA. Vol. 4098 of Lecture Notes in Computer Science. Springer, pp. 242–256.

Armando, A., Rusinowitch, M., Stratulat, S., 2002. Incorporating decision procedures in implicit induction. J. Symb. Comput. 34 (4), 241–258, a previous version appeared in Calculemus 2001 (9th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning), pages 65-75.

Baader, F., Nipkow, T., 1998. Term Rewriting and All That. Cambridge University Press.

Bachmair, L., Ganzinger, H., 2001. Resolution theorem proving. In: Robinson, A., Voronkov, A. (Eds.), Handbook of Automated Reasoning. Elsevier and MIT Press, pp. 19–99.

Balaa, A., Bertot, Y., 2000. Fix-point equations for well-founded recursion in type theory. In: Aagaard, M., Harrison, J. (Eds.), Theorem Proving in Higher Order Logics. Vol. 1869 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 1–16.

Barthe, G., Courtieu, P., 2002. Efficient reasoning about executable specifications in Coq. In: Theorem Proving in Higher Order Logics. Vol. 2410 of LNCS. Springer Berlin, pp. 31–46.

Barthe, G., Forest, J., Pichardie, D., Rusu, V., 2006. Defining and reasoning about recursive functions: A practical tool for the Coq proof assistant. In: Hagiya, M., Wadler, P. (Eds.), Functional and Logic Programming. Vol. 3945 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 114–129.

Barthe, G., Stratulat, S., 2003. Validation of the JavaCard platform with implicit induction techniques. In: Nieuwenhuis, R. (Ed.), RTA. Vol. 2706 of Lecture Notes in Computer Science. Springer, pp. 337–351.

Berregeb, N., Bouhoula, A., Rusinowitch, M., 1996. SPIKE-AC: A system for proofs by induction in associative-commutative theories. In: Ganzinger, H. (Ed.), Rewriting Techniques and Applications. Vol. 1103 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 428–431.

Berregeb, N., Bouhoula, A., Rusinowitch, M., 1998. Observational proofs with critical contexts. In: Fundamental Approaches to Software Engineering. pp. 38–53.
URL http://dx.doi.org/10.1007/BFb0053582

Blanqui, F., Koprowski, A., 2011. CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. MSCS 21 (4), 827–859.

Bouhoula, A., 1994. SPIKE: A system for sufficient completeness and parameterized inductive proofs. In: Bundy, A. (Ed.), Automated Deduction — CADE-12. Vol. 814 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 836–840.

Bouhoula, A., 1996. Using induction and rewriting to verify and complete parameterized specifications. Theoretical Computer Science 170, 170–1.

Bouhoula, A., 2009. Simultaneous checking of completeness and ground confluence for algebraic specifications. ACM Transactions on Computational Logic (TOCL) 10 (3), 1–33.

Bouhoula, A., Jouannaud, J., 2001. Automata-Driven Automated Induction. Information and Computation 169 (1), 1–22.

Bouhoula, A., Kounalis, E., Rusinowitch, M., 1992. SPIKE, an automatic theorem prover. In: Logic Programming and Automated Reasoning (LPAR). pp. 460–462.

Bouhoula, A., Kounalis, E., Rusinowitch, M., 1995. Automated mathematical induction. Journal of Logic and Computation 5 (5), 631–668.

Bouhoula, A., Rusinowitch, M., 2002. Observational proofs by rewriting. Theoretical Computer Science 275 (1–2), 675–698.
URL http://citeseer.ist.psu.edu/451958.html

Boulton, R., Slind, K., 2000. Automatic derivation and application of induction schemes for mutually recursive functions. In: Lloyd, J., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Palamidessi, C., Pereira, L., Sagiv, Y., Stuckey, P. (Eds.), Computational Logic — CL 2000. Vol. 1861 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 629–643.

Boyer, R., Moore, J. S., 1991. MJRTY—A Fast Majority Vote Algorithm. Automated Reasoning: Essays in Honor of Woody Bledsoe.

Boyer, R. S., Moore, J. S., 1988. A computational logic handbook. Academic Press Professional.

Bronsard, F., Reddy, U., Hasker, R., 1994. Induction using term orderings. In: Automated Deduction —CADE-12. Vol. 814 of LNCS. Springer, pp. 102–117.

Bronsard, F., Reddy, U. S., 1991. Conditional rewriting in Focus. In: Conditional and Typed Rewriting Systems. pp. 1–13.

Brotherston, J., Simpson, A., 2011. Sequent calculi for induction and infinite descent. J. Log. Comput. 21 (6), 1177–1216.

Comon, H., 2001. Inductionless induction. In: Robinson, A., Voronkov, A. (Eds.), Handbook of Automated Reasoning. Elsevier and MIT Press, pp. 913–962.

Contejean, E., Courtieu, P., Forest, J., Pons, O., Urbain, X., 2007. Certification of automated termination proofs. Frontiers of Combining Systems, 148–162.

Contejean, E., Paskevich, A., Urbain, X., Courtieu, P., Pons, O., Forest, J., 2010. A3PAT, an approach for certified automated termination proofs. In: Gallagher, J. P., Voigtländer, J. (Eds.), PEPM - Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010, Madrid, Spain, January 18-19, 2010. ACM, pp. 63–72.

Courant, J., 1996. Proof reconstruction. Research Report RR96-26, LIP, preliminary version.

Deplagne, E., Kirchner, C., Kirchner, H., Nguyen, Q. H., 2003. Proof search and proof check for equational and inductive theorems. In: Automated Deduction – CADE-19. No. 2741 in Lecture Notes Computer Science. pp. 297–316.

Dershowitz, N., 1982. Orderings for term-rewriting systems. Theoretical Computer Science 17 (3), 279–301.

Dershowitz, N., Reddy, U. S., 1993. Deductive and inductive synthesis of equational programs. Journal of Symbolic Computation 15 (5/6), 467–494.

Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Roux, S. L., Mahboubi, A., O'Connor, R., Biha, S. O., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., Théry, L., 2013. A machine-checked proof of the Odd Order Theorem. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (Eds.), Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings. Vol. 7998 of Lecture Notes Computer Science. Springer, pp. 163–179.

Henaien, A., Stratulat, S., 2013. Performing implicit induction reasoning with certifying proof environments. In: Bouhoula, A., Ida, T., Kamareddine, F. (Eds.), Proceedings Fourth International Symposium on Symbolic Computation in Software Science, Gammarth, Tunisia, 15-17 December 2012. Vol. 122 of Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, pp. 97–108.

Howe, D. J., 1993. Reasoning about functional programs in Nuprl. In: Functional Programming, Concurrency, Simulation and Automated Reasoning. Vol. 693 of Lecture Notes in Computer Science. Springer Verlag, pp. 145–164.

Kaliszyk, C., 2005. Validation des preuves par récurrence implicite avec des outils basés sur le calcul des constructions inductives. Master's thesis, Université Paul Verlaine - Metz.

Kamin, S., Lévy, J. J., 1980. Two generalizations of the recursive path ordering, unpublished manuscript, University of Illinois, IL, USA.

Kapur, D., Musser, D. R., 1987. Proof by consistency. Artificial Intelligence 31 (2), 125–157.

Kapur, D., Narendran, P., Zhang, H., 1986. Proof by induction using test sets. In: 8th International Conference on Automated Deduction. Vol. 230 of Lecture Notes Computer Science. Springer, pp. 99–117.

Kapur, D., Subramaniam, M., 1996. Automating induction over mutually recursive functions. In: Algebraic Methodology and Software Technology. Vol. 1101 of LNCS. Springer, pp. 117–131.

Kapur, D., Zhang, H., 1988. RRL: A rewrite rule laboratory. In: Lusk, E., Overbeek, R. (Eds.), 9th International Conference on Automated Deduction. Vol. 310 of LNCS. Springer Berlin / Heidelberg, pp. 768–769.

Kounalis, E., Rusinowitch, M., 1990. Mechanizing inductive reasoning. In: Proceedings of the eighth National conference on Artificial intelligence - Volume 1. AAAI'90. AAAI Press, pp. 240–245.

Lescanne, P., 1983. Computer experiments with the REVE term rewriting system generator. In: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. POPL '83. ACM, New York, NY, USA, pp. 99–108.
URL http://doi.acm.org/10.1145/567067.567078

Lescanne, P., 1990. Implementation of completion by transition rules + control: ORME. In: Kirchner, H., Wechler, W. (Eds.), Algebraic and Logic Programming. Vol. 463 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 262–269.
URL http://dx.doi.org/10.1007/3-540-53162-9_44

Liu, P., Chang, R.-J., 1987. A new structural induction scheme for proving properties of mutually recursive concepts. In: 6th National Conference on Artificial Intelligence. Vol. 1. AAAI Press, pp. 144–148.
URL http://www.informatik.uni-trier.de/~ley/db/conf/aaai/aaai87.html#LiuC87

Musser, D. R., 1980. On proving inductive properties of abstract data types. In: POPL. pp. 154–162.

Nieuwenhuis, R., Rubio, A., 2001. Paramodulation-based theorem proving. In: Robinson, A., Voronkov, A. (Eds.), Handbook of Automated Reasoning. Elsevier and MIT Press, pp. 371–443.

Nipkow, T., Paulson, L. C., Wenzel, M., 2002. Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Vol. 2283 of Lecture Notes in Computer Science. Springer.

Protzen, M., 1994. Lazy generation of induction hypotheses. Automated Deduction — CADE-12, 42–56.

Reddy, U., 1990. Term Rewriting Induction. Proceedings of the 10th International Conference on Automated Deduction, 162–177.

Rusinowitch, M., Stratulat, S., Klay, F., 2003. Mechanical verification of an ideal incremental ABR conformance algorithm. J. Autom. Reasoning 30 (2), 53–177.

Sozeau, M., 2007. Program-ing finger trees in Coq. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming. ICFP '07. ACM, New York, NY, USA, pp. 13–24.
URL http://doi.acm.org/10.1145/1291151.1291156

Sozeau, M., 2010. Equations: A dependent pattern-matching compiler. In: Kaufmann, M., Paulson, L. C. (Eds.), Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings. Vol. 6172 of Lecture Notes Computer Science. Springer, pp. 419–434.

Stratulat, S., November 2000. Preuves par récurrence avec ensembles couvrants contextuels. applications à la vérification de logiciels de télécommunications. Ph.D. thesis, Université Henri Poincaré, Nancy I, also published as a book at 'Editions Universitaires Européennes' in 2012, ISBN 978-3841794901.

Stratulat, S., 2001. A general framework to build contextual cover set induction provers. J. Symb. Comput. 32 (4), 403–445.

Stratulat, S., 2008. Combining rewriting with Noetherian induction to reason on non-orientable equalities. In: Voronkov, A. (Ed.), Rewriting Techniques and Applications. Vol. 5117 of Lecture Notes in Computer Science. Springer Berlin, pp. 351–365.

Stratulat, S., 2010. Integrating implicit induction proofs into certified proof environments. In: IFM'2010 (8th International Conference on Integrated Formal Methods). Vol. 6396 of Lecture Notes in Computer Science. pp. 320–335.

Stratulat, S., 2012. A unified view of induction reasoning for first-order logic. In: Voronkov, A. (Ed.), Turing-100 (The Alan Turing Centenary Conference). Vol. 10 of EPiC Series. EasyChair, pp. 326–352.

Stratulat, S., 2014. Implementing reasoning modules in implicit induction theorem provers. In: SYNASC 2014: Proceedings of the 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. IEEE, pp. 133–140.

Stratulat, S., Demange, V., 2011. Automated certification of implicit induction proofs. In: CPP'2011 (First International Conference on Certified Programs and Proofs). Vol. 7086 of Lecture Notes Computer Science. Springer Verlag, pp. 37–53.

The CompCert project, 2014.
URL http://compcert.inria.fr/

The Coq development team, 2013. The Coq Reference Manual - version 8.4. INRIA.
URL http://coq.inria.fr/doc

The Flyspeck website, 2014.
URL https://code.google.com/p/flyspeck/

The L4.verified project, 2014.
URL http://www.ertos.nicta.com.au/research/l4.verified/

The SPIKE development team, 2014. The SPIKE prover. https://github.com/sorinica/spike-prover.

Voicu, R., Li, M., 2009. Descente Infinie proofs in Coq. In: The 1st Coq Workshop, München (Germany).

Walther, C., 1993. Combining induction axioms by machine. In: Bajcsy, R. (Ed.), IJCAI - Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993. Morgan Kaufmann, pp. 95–101.

Wirth, C.-P., 2004. Descente infinie + Deduction. Logic Journal of the IGPL 12 (1), 1–96.

A. Coq script and SPIKE specification

A.1. The 'P and Q' proof script in Coq

(* importing the COCCINELLE term algebra and the defined tactics for automatic ordering reasoning *)

```
Require Import pandq_predicate_spec.
Definition type_LF_PandQ := nat → nat → Prop × List.list term.

Definition Pu : type_LF_PandQ :=
  fun u _ ⇒ (P u, (model_nat u :: model_nat u :: nil)).
Definition Qxy : type_LF_PandQ :=
  fun x y ⇒ (Q x y, (model_nat x :: model_nat x :: model_nat y :: nil)).
Definition LF_PandQ := [Pu, Qxy].
Fixpoint f_P (u1 : nat): bool := match u1 with
                                 | 0 ⇒ true
                                 | S u2 ⇒ true
                                 end.
Functional Scheme f_P_ind := Induction for f_P Sort Prop.
Fixpoint f_Q (u2 : nat) (u1 : nat) {struct u2}: bool :=
  match u2, u1 with
  | 0, _ ⇒ true
  | S u4, _ ⇒ true
  end.
Functional Scheme f_Q_ind := Induction for f_Q Sort Prop.
Lemma main_PandQ :
 ∀ F,
 In F LF_PandQ →
 ∀ u1 u2,
 (∀ F',
  In F' LF_PandQ →
  ∀ e1 e2, less (snd (F' e1 e2)) (snd (F u1 u2)) → fst (F' e1 e2)) →
 fst (F u1 u2).
Proof.

(* the case analysis on the content of LF *)
intros F HF u1 u2; case_In HF; intro Hind.

(* split on P *)
rename u1 into _u1. rename u2 into d_u2.
rename _u1 into u1. revert Hind.
pattern u1, (f_P u1). apply f_P_ind.
(* case 0 *)
intros _u1. intro. intro HFabs0.
unfold Pu. unfold fst. apply p0.
```

```
(* case S *)
intros _u2 _u1. intro. intro HFabs0.
unfold Pu. unfold fst.
apply p1.
(* applying Pu as IH *)
pose proof (HFabs0 Pu) as Hind. clear HFabs0.
assert (fst (Pu _u1 0)) as HFabs0.
apply Hind. trivial_in 0.
unfold fst. unfold Pu.
unfold snd. rewrite_model. abstract solve_rpo_mul. (* solving the ordering part *)
unfold fst in HFabs0. unfold Pu in HFabs0. trivial. (* solving the deduction part *)
(* applying Qxy as IH *)
pose proof (HFabs0 Qxy) as Hind. clear HFabs0.
assert (fst (Qxy _u1 (S _u1))) as HFabs0.
apply Hind. trivial_in 1.
unfold fst. unfold Qxy. unfold Pu.
unfold snd. rewrite_model. abstract solve_rpo_mul. (* solving the ordering part *)
unfold fst in HFabs0. unfold Qxy in HFabs0. trivial.(* solving the deduction part *)

(* split on Q *)
rename u1 into _u1. rename u2 into _u2.
rename _u1 into u1. rename _u2 into u2. revert Hind.
pattern u2, u1, (f_Q u2 u1). apply f_Q_ind.
(* case 0 *)
intros.
unfold Qxy. unfold fst. apply q0.
(* case S *)
intros _u2 _u1 u3 e. intro HFabs0.
unfold Qxy. unfold fst. apply q1.
(* applying Qxy as IH *)
pose proof (HFabs0 Qxy) as Hind. clear HFabs0.
assert (fst (Qxy _u1 u3)) as HFabs0.
apply Hind. trivial_in 1.
unfold fst. unfold Qxy.
unfold snd. rewrite_model. abstract solve_rpo_mul. (* solving the ordering part *)
unfold fst in HFabs0. unfold Qxy in HFabs0. trivial.(* solving the deduction part *)
(* applying Pu as IH *)
pose proof (HFabs0 Pu) as Hind. clear HFabs0.
assert (fst (Pu _u1 0)) as HFabs0.
apply Hind. trivial_in 0.
unfold fst. unfold Pu. unfold Qxy.
unfold snd. rewrite_model. abstract solve_rpo_mul. (* solving the ordering part *)
unfold fst in HFabs0. unfold Pu in HFabs0. trivial.(* solving the deduction part *)

Qed.

Definition S_PandQ :=
  fun f ⇒ ∃ F, In F LF_PandQ ∧ (∃ e1, ∃ e2, f = F e1 e2).
```

```
Theorem all_true_PandQ :
 ∀ F, In F LF_PandQ → ∀ (u1 : nat) (u2 : nat), fst (F u1 u2).
Proof.
(* automatically generated *)
Qed.

Theorem true_Qxy : ∀ (x : nat) (y : nat), Q x y.
Proof.
do 2 intro.
apply (all_true_PandQ Qxy); trivial_in 1 || (repeat constructor).
Qed.
```

A.2. The SPIKE specification of the 'even' and 'odd' example

```
specification : even

sorts : nat  bool;

$
$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.
$

constructors :

0    :          -> nat;
S_ : nat -> nat;
true : bool;
false : bool;

defined functions:

even_ : nat -> bool;
odd_ : nat -> bool;
_+_ : nat nat -> nat;

axioms:

0 + x = x;
```

```
S(x) +  y = S(x + y);

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

even (0) = true;
even (S(x))  = odd (x);

odd(0) =false;
odd (S(x))  = even (x);

$
$Fixpoint even (x: nat): bool :=
$match x with
$| 0 => true
$| (S x') => odd x'
$end
$with
$odd (x: nat): bool :=
$match x with
$| 0 => false
$| (S x') => even x'
$end.
$

greater:
even : + S 0 true false;
+ : S 0;

equiv:
even odd;

properties:

system_is_sufficiently_complete ;
system_is_ground_convergent ;

Strategy:

% instances of Delete
tautology_rule                  = delete(id, [tautology]);
subsumption_rule                = delete(id, [subsumption (L|C)]);
```

```
negative_clash_rule              = Delete(id, [negative_clash]);

% instances of Simplify
rewriting_rule = simplify(id,[rewriting(rewrite, L|R|C, *)]);

% instances of AddPremise
inst_var_rule = add_premise(generate,[id]);


stra = repeat (try (
                    tautology_rule,
                    negative_clash_rule,
                    subsumption_rule,
                    rewriting_rule,
                    print_goals_with_history(t)
));

fullind = (repeat(stra, inst_var_rule), print_goals_with_history);

start_with: fullind

conjectures:

odd (x + y) = true, even (y + z) = true => odd (x + z) = true ;
```